André Caldeira Pereira

Licenciado

Conversão Automática de Código CLP para Ambientes de Simulação com Recurso a Normas Internacionais

Dissertação para obtenção do Grau de Mestre em Engenharia Electrotécnica e de Computadores

Orientador: João Francisco Alves Martins, Professor Auxiliar, Universidade Nova de Lisboa
Coorientador: Celson Pantoja Lima, Professor Auxiliar, Universidade Nova de Lisboa

Júri:

Presidente: Prof. Doutor Luís Filipe dos Santos Gomes
Vogal: Prof. Doutor Luís Filipe Figueira de Brito Palma
Vogal: Prof. Doutor João Francisco Alves Martins
Vogal: Prof. Doutor Celson Pantoja Lima

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Novembro 2011

INDICAÇÃO DE COPYRIGHT

Dedico esta dissertação ao meu irmão Vasco, que faleceu no início do meu curso, mas cujos ensinamentos para com a vida entretanto me deram forças para me tornar uma pessoa melhor.

AGRADECIMENTOS

Aos meus pais, sem o seu apoio nunca teria ingressado no ensino superior e chegado onde cheguei, nem seria a pessoa que sou.

Ao meu avô, por me ter possibilitado concluir o curso e apoiado quando precisei.

Aos meus colegas, que tornaram o martírio da faculdade numa experiência edificante e divertida, para além de ainda fazerem revisão à minha dissertação.

À Sara, por me ter apoiado, aconselhado e revisto o meu trabalho desta dissertação.

Aos meus orientadores de dissertação, que tiveram um papel pessoal diferente para mim: o professor Celson Lima, com o qual aprendi a exigir mais de mim mesmo e alcançar mais longe; o professor João Martins que me mostrou que era possível aprender eletrotecnia com paixão pelo tema.

Agradeço ainda ao Husam Aldahiyat pela sua permissão em utilizar a biblioteca "goto.m" na minha dissertação.

RESUMO

Este trabalho visa possibilitar o controlo por CLP (Controlador Lógico Programável) de um modelo de processo em ambiente de simulação. O trabalho desenvolvido recorre à norma IEC (*International Electrotechnical Commission*) 61131-3, com destaque para a linguagem IL (*Instruction List)*.

Foi desenvolvido um conversor, de nome **Matlaber,** que recebe código CLP em linguagem IL e o converte automaticamente para um ficheiro ".m", compatível com o ambiente *Matlab*/*Simulink*. Este ficheiro será uma função *Matlab* que simula um CLP correndo um programa equivalente ao original, para uso no ambiente *Simulink*.

O trabalho desenvolvido inclui também um tradutor de códigos CLP proprietários, de nome **UnifIL**, que normaliza o código CLP proprietário em código IL.

Com estes dois elementos, torna-se possível emular o funcionamento de programas de controlo de CLP proprietários, em ambiente *Matlab*/*Simulink*.

A tradução do código proprietário é feita com recurso a dicionários de regras e tradução próprios, sendo portanto expansível a várias linguagens de CLP de baixo nível, desde que um dicionário adequado seja criado. Para a prova de conceito foram criados 2 dicionários para linguagens proprietárias: *Siemens Step 7 (*STL série 200*)* e *Mitsubishi Q Series* (*List Mode*).

Palavras-chave: CLP, Matlab, Simulink, programação, proprietário, norma, IEC 61131, Instruction List, simulação, conversão, tradução, Matlaber, UnifIL.

ABSTRACT

The aim of the presented work is to enable PLC (Programmable Logic Computer) control of a process model within a simulation environment. The work is based on the IEC (International Electrotechnical Commission) 61131-3 standard, directed to the IL (Instruction List) programming language.

The developed conversion tool, named **Matlaber**, receives PLC code in IL language and converts it automatically to an ".m" file, compatible with the Matlab/Simulink environment. This file will be a Matlab function which simulates a PLC by running a program equivalent to the original one, for use in the Simulink environment.

Another tool was also developed, a proprietary PLC code translator, named **UnifIL**, which standardizes proprietary PLC code to IL language code.

With these two tools, it's possible to emulate to behaviour of proprietary PLC control programs, within the Matlab/Simulink environment.

The proprietary code translation is processed by resourcing to specific, language directed, translation rules dictionaries, as to be expandable to any low level PLC programming language. For the developed work and as proof of concept, 2 different translation dictionaries were created, for the following proprietary languages: Siemens Step 7 (STL 200 series) and Mitsubishi Q Series (List Mode).

Keywords: PLC, Matlab, Simulink, programming, proprietary, standard, IEC 61131, Instruction List, simulation, conversion, translation, Matlaber, UnifIL.

LISTA DE SIGLAS E ACRÓNIMOS

CLP – Controlador Lógico Programável

FBD – *Function Block Diagram*

IHM – Interface Homem Máquina

I/O – *Input/Output*

IEC – *International Electrotechnical Commission*

IDE – *Integrated Development Environment*

IL – *Instruction List*

LED – *Light Emitting Diode*

LD – *Ladder Diagram*

PID – Proporcional, Integral e Derivada

POU – *Program Organization Unit*

SCAD – Supervisão, Controlo e Aquisição de Dados

SFC – *Sequential Function Chart*

ST – *Structured Text*

STL – *Statement List*

XML – *eXtensible Markup Language*

ÍNDICE DE MATÉRIAS

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

1 INTRODUÇÃO

O conceito da automação pode-se resumir na execução automatizada de tarefas, sem intervenção humana, na produção de bens e serviços. Com o advento das primeiras aplicações elétricas, a automação criou uma mudança de paradigma ao substituir operários em tarefas repetitivas, ou de grande carga física, por autómatos [1]. Após a revolução industrial, que teve um grande papel na economia mundial, a automação industrial foi o passo seguinte na evolução dos ambientes fabris, Na atualidade, são utilizados Controladores Lógicos Programáveis (conhecidos simplesmente pela sigla CLP) para controlar os mais variados processos fabris. De uma forma simples, pode-se afirmar que os CLP são controladores programáveis com múltiplas entradas e saídas. Adequados ao ambiente fabril do mundo da automação são extremamente robustos e fiáveis.

1.1 O PROBLEMA

O uso dos CLP para controlo de processos industriais não é livre de problemas, designadamente no que concerne ao desenvolvimento dos programas de controlo. Um dos grandes problemas diz respeito aos testes associados ao desenvolvimento de um programa de controlo, pois os processos são, geralmente, grandes e complexos com muitas entradas e saídas. Um pequeno erro pode comprometer toda a linha de produção, sendo por isso essencial testar, validar e analisar programas de controlo de CLP.

Para testar programas de CLP, existem vários métodos que podem ser utilizados [2][3][4]:

- Modelos à escala.
- Conjuntos de indicadores luminosos (tipicamente LED – *Light Emitting Diode*) e interruptores.
- Interfaces Homem Máquina (IHM).
- Sistemas de Supervisão, Controlo e Aquisição de Dados (SCAD).
- Ferramentas de simulação baseados em computador (exploradas no capítulo 3.2).

Cada uma destas abordagens apresenta as suas vantagens e desvantagens. O uso de indicadores luminosos e interruptores é extremamente confuso e pouco apelativo, sendo apenas útil em processos de reduzida dimensão. Os sistemas IHM e SCAD, em alguns casos, permitem tais

funcionalidades mas não foram desenvolvidos para este propósito, têm um custo associado elevado e normalmente apenas funcionam com protocolos proprietários, o que limita a sua abrangência. O uso de modelos à escala dos processos reais (como o exemplo na Figura 1.1) é uma forma fidedigna forma de testar o programa, permitindo a validação e verificação de resultados em ambiente quase real.

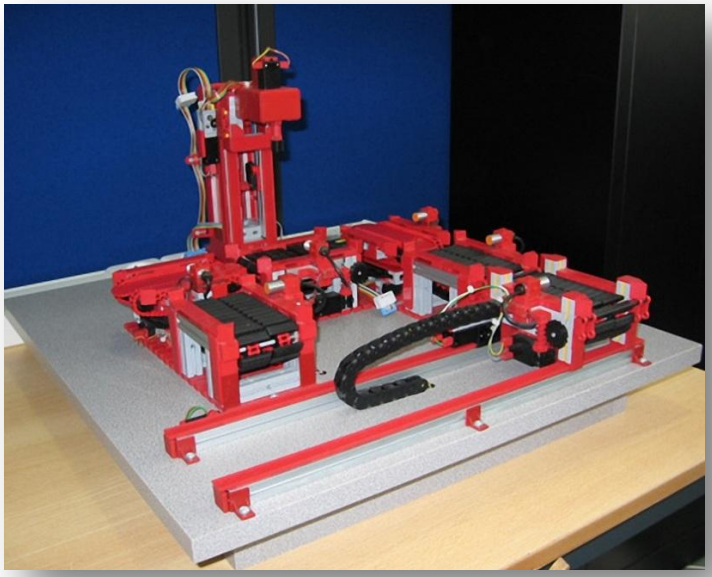

Figura 1.1 - Exemplo de modelo à escala[1] [5]

Torna-se, no entanto, dispendioso e difícil de adaptar a diferentes processos, particularmente no caso de processos industriais complexos.

Foram ainda desenvolvidas algumas ferramentas de simulação de processos em computador, usando tecnologias baseadas em microcontroladores e desenhadas para funcionar com qualquer tipo de CLP [6]. Também estão disponíveis ferramentas comerciais de simulação como *PC-SIM* [7] ou o *PSIM* [8]. No entanto, nenhuma destas ferramentas se adequa a processos complexos de controlo realimentado, estando limitados aos seus modelos internos e/ou às ferramentas internas de modelação (exemplo da Figura 1.2). O seu alcance torna-se bastante limitado ao não permitir integração com outras ferramentas de simulação de processos industriais.

[1] Modelo *Stamper* [5], da *Staudinger* utilizado nos laboratórios de automação da FCT.

No campo da educação, questões de custo e motivação tornam-se ainda mais importantes, sendo as soluções disponíveis opostas em termos de custo face à motivação: um conjunto de LEDs e interruptores é barato e pouco educativo, mas um modelo à escala é dispendioso mas motivador.

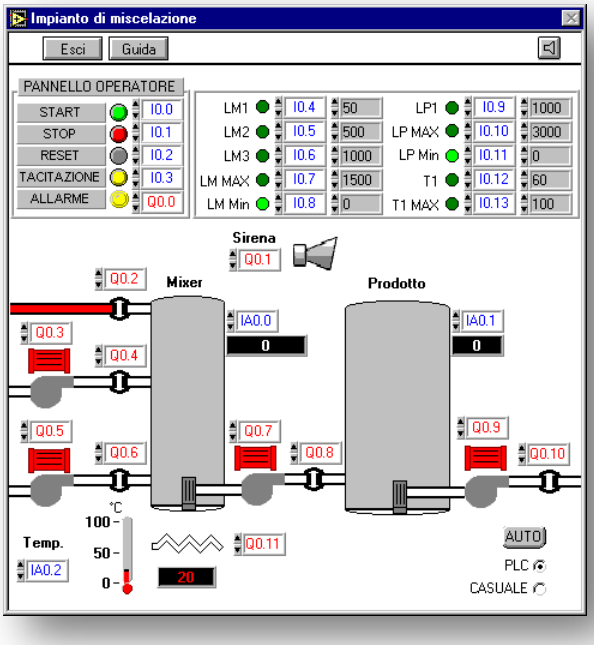

Figura 1.2 – Modelo fabril com dois tanques implementado no ambiente *PC-Sim* [7]

Do anteriormente exposto, pode concluir-se que existe uma lacuna no binómio desenvolvimento/testes ao programa. Com o trabalho desenvolvido propõe-se preencher essa lacuna.

1.2 SOLUÇÃO PROPOSTA

Numa indústria onde um grande problema são os testes associados ao desenvolvimento de um programa de controlo, o trabalho desenvolvido pretende possibilitar a análise, teste e validação do programa de controlo que será imposto no CLP. A solução proposta assume a existência de um modelo do processo em estudo ou possibilidade dessa modelação, com todas as vantagens que daí advêm [9]. Na Figura 1.3 é possível discernir mais claramente o problema.

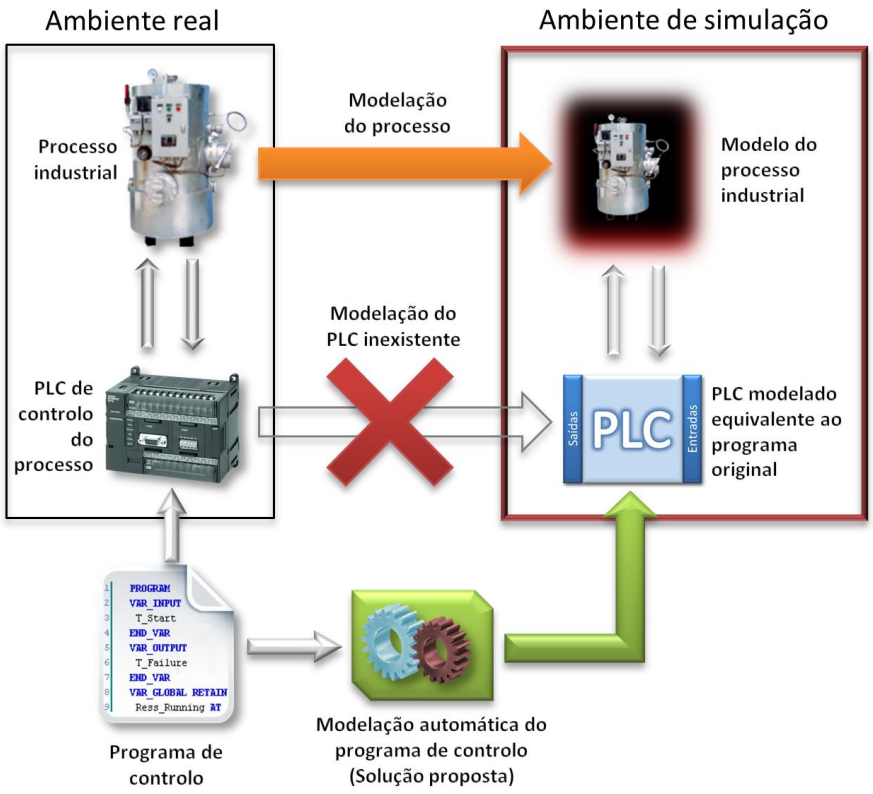

Figura 1.3 – Visão geral do problema e da solução proposta

Do lado esquerdo da Figura 1.3 está ilustrado o ambiente real, com o processo industrial e o CLP de controlo, o qual recebe o seu programa de controlo sob a forma de um ficheiro de texto. No campo da simulação, temos o modelo equivalente ao processo industrial, criado através de modelação de processos, juntamente com o CLP em ambiente de simulação. A "**Modelação do CLP inexistente**" é onde se situa a referida lacuna de desenvolvimento atual [6][10]. A solução proposta, "**Modelação automática do programa de controlo**", será um sistema computacional que receba, como entrada, o ficheiro de texto do programa de controlo e gere um CLP simulado compatível com o ambiente de simulação. Para efeitos de controlo, o comportamento do CLP simulado será equivalente ao CLP real. Ao juntar o CLP simulado ao modelo do processo industrial, torna-se possível simular fielmente o controlo desse processo, recorrendo ao mesmo programa de controlo que será utilizado no CLP real, permitindo verificação de resultados simulados. O desenvolvimento do programa poderá continuar a ser feito no IDE (*Integrated Development Environment*) do fabricante, retendo todas particularidades do projeto em dado *hardware*/*software*.

O ambiente de simulação alvo será o *Matlab* [11], com ênfase no ambiente *Simulink*. O *Matlab* é a ferramenta de eleição no que toca a simulações, modelação e análise. Juntamente com a sua

versatilidade e capacidade de integrar outras ferramentas de modelação, o *Matlab* é ideal para a solução proposta.

O programa de controlo poderá ser código CLP de acordo com a norma IEC 61131-3 [12][13], especificamente na linguagem IL (*Instruction List*). A norma tem tido um apoio crescente dos fabricantes, tornando-se o alvo ideal para uma solução como a apresentada.

Para além do suporte à norma, programas de controlo de CLP proprietários de variadas marcas, poderão ser suportados através de outra ferramenta, desenvolvida neste trabalho. A esta ferramenta incube-se a tarefa de transformar código proprietário em código normalizado, aceite pelo sistema computacional de modelação dos CLP.

1.3 OBJETIVOS

O objetivo principal do trabalho apresentado é de possibilitar simulações de processos controlados por CLP no ambiente *Matlab*/*Simulink*, dado o programa de CLP de controlo original. Para realizar esse objetivo, são necessários 2 processos complementares.

Um processo deverá permitir simular, em ambiente *Matlab*/*Simulink*, o funcionamento de um dado programa de CLP em código IL da norma IEC 61131-3. Para tornar isto possível, é necessário desenvolver uma ferramenta computacional de conversão de código IL para o ambiente *Matlab*/*Simulink*.

Para casos de programas de controlo originais em códigos proprietários, outro processo deverá permitir normalizar, para IL, códigos CLP de linguagens proprietárias, através do desenvolvimento de uma ferramenta computacional de tradução.

1.4 ESTRUTURA DO DOCUMENTO

A presente dissertação encontra-se estruturada em 6 capítulos:

1. **ENQUADRAMENTO DOS CLP NA AUTOMAÇÃO INDUSTRIAL** – Breve descrição do funcionamento geral e história dos CLP, para além de um enquadramento da norma IEC 61131.
2. **SIMULAÇÃO DE PROCESSOS** – Descrição e análise das várias abordagens computadorizadas, já existentes, de simulação e teste de processos controlados por CLP.
3. **SOLUÇÃO CONCEPTUAL** – É apresentada a solução proposta sob a forma de um modelo conceptual.
4. **IMPLEMENTAÇÃO** – Descreve a implementação do trabalho, desenvolvido na solução conceptual.
5. **EXPERIÊNCIAS DE VALIDAÇÃO** – Expõe as várias experiências que foram efetuadas para validação do trabalho desenvolvido.
6. **CONCLUSÕES E TRABALHO FUTURO** – Reflexão sobre o trabalho desenvolvido e trabalho futuro.

2 ENQUADRAMENTO DOS CLP NA AUTOMAÇÃO INDUSTRIAL

Neste capítulo enquadra-se os CLP no ambiente da automação industrial, fazendo uma breve descrição seu funcionamento geral e história, para além de um enquadramento da norma vigente no desenvolvimento de CLP – o IEC 61131.

2.1 CLP

As primeiras aplicações de CLP surgiram na década de 1960, como resposta aos custos associados ao controlo por sistemas de relés inflexíveis. Tiveram um início humilde, sendo inicialmente desenhados como substitutos de arranjos de relés de controlo. A sua flexibilidade trazia vantagens enormes, substituindo rapidamente o uso de controlo por sistemas de relés. Entre essas vantagens inclui-se: diminuição de custos, aumento de funcionalidades e redução da sua pegada espacial e energética [1]. Na Figura 2.1 é visível um exemplo de um micro CLP.

Figura 2.1 – Exemplo de um micro CLP[2]

[2] Modelo **CP1L** da **Omron, lançado em 2007.**

Atualmente, os CLP modernos foram já alvo de vários avanços tecnológicos, entre eles:

- Velocidades de leitura e de resposta elevadas.
- Modelos com alta densidade de entradas e saídas.
- Interfaces inteligentes que permitem funcionalidades de ligação avançadas como processamento distribuído, controladores PID (ação Proporcional, Integral e Derivada) e ligações a vários tipos de redes.
- Integração de terminal de ligações físicas numa única unidade.
- Possibilidade de ligação direta a dispositivos de medida, como termopares e extensómetros.

Existe uma grande variedade de modelos cuja escala varia consoante o número de entradas/saídas, desde modelos com 10 (Figura 2.1), até modelos com 8000.

De uma forma simples, os CLP podem ser descritos como controladores programáveis industriais de estado sólido, usando circuitos integrados para implementar funções de controlo eléctrico. A Figura 2.2 mostra um diagrama conceptual de um CLP típico. Um CLP pode ser conceptualmente dividido em memória, unidade lógica e I/O (interface de entradas e saídas).

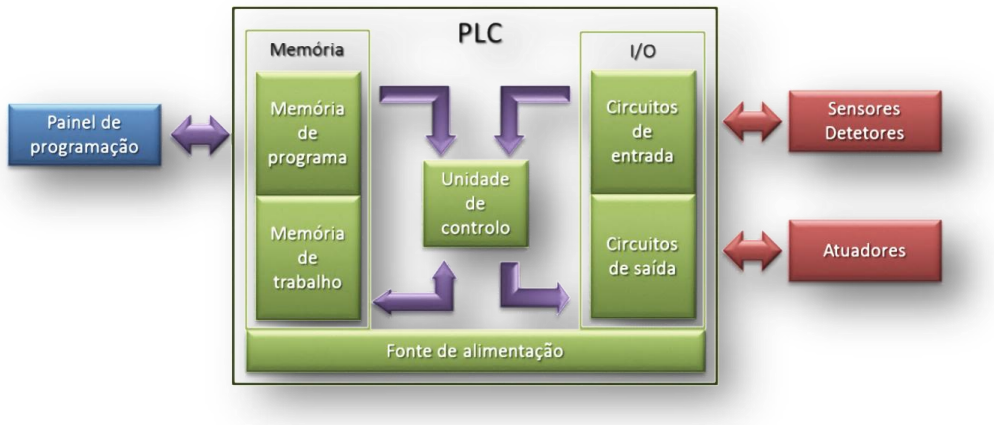

Figura 2.2 - Diagrama conceptual de um CLP típico

A memória inclui a unidade volátil e não volátil, sendo esta última o local de armazenamento do programa de controlo. Alguns dados (valores de memória especiais – de retenção) são mantidos mesmo após o CLP ter sido desligado.

A unidade lógica consiste num numa unidade controlo, semelhante a um processador central, que é responsável pela interpretação das várias funções e instruções, assim como operações

matemáticas e de lógica [14]. A interface é responsável pela gestão das múltiplas entradas e saídas, de leitura e de controlo que, num CLP mais complexo, poderão chegar aos milhares.

O funcionamento de um CLP é cíclico e baseado num sinal de relógio. Entre cada ciclo de funcionamento existem, basicamente, 3 passos que se repetem: aquisição de entradas, processamento e geração de saídas, como ilustrado na Figura 2.3.

Figura 2.3 - Funcionamento cíclico de um CLP

A aquisição de entradas implica escrever para memória os valores dos sensores e sinais à entrada, para que no passo do processamento, onde o programa de controlo é executado, possa escrever para os valores de memória de saída o resultados das operações de comando. Finalmente, aquando da geração de saídas, os valores em memória são escritos para as saídas físicas do CLP.

No contexto de controlo de um processo industrial, as saídas e entradas correspondem, respetivamente, às ações de controlo do processo e leituras de sensores (ver Figura 2.4). O CLP lê os valores dos sensores e interruptores, processa o código do seu programa de controlo e gera as respetivas saídas e ações de controlo (o funcionamento cíclico apresentado na Figura 2.3).

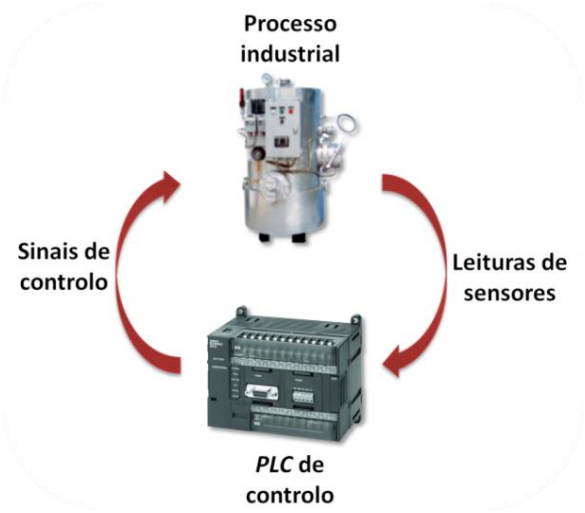

Figura 2.4 - Funcionamento de um CLP no contexto de um processo industrial

2.2 A NORMA IEC 61131-3

Em 1992, o *International Electrotechnical Commission* [15] (conhecido pela sigla IEC) publicou a norma IEC 1131. O objetivo era o de harmonizar a indústria dos CLP sob uma plataforma comum de desenvolvimento, tanto de *software* como de *hardware*. Devido a questões de consistência de nomenclatura e colisão com organismos de alguns países, foi renomeada para IEC 61131, como é atualmente conhecida.

As diferentes partes do IEC 61131 são [13][16]:

- Parte 1: uma visão geral com definições.
- Parte 2: requisitos e testes para os equipamentos.
- Parte 3: linguagens de programação.
- Parte 4: guia de implementação para utilizadores.
- Parte 5: especificações para a comunicação.
- Parte 6: segurança funcional em CLP.
- Parte 7: programação com controlo difuso.
- Parte 8: guias de aplicação e implementação das linguagens de programação.

Para o presente trabalho, a **parte 3** da norma é a mais relevante. Conhecida simplesmente como IEC 61131-3, define um modelo de desenvolvimento comum de programação de CLP, onde se incluem as 5 linguagens de programação. As 5 linguagens disponíveis e sua descrição, podem ser consultadas na Tabela 2.1.

Tabela 2.1 - As linguagens suportadas pelo IEC 61131-3 [13]

Linguagem	Descrição
SFC	*Sequential Function Chart*. Linguagem gráfica que descreve claramente o andamento do programa, definindo que ações do processo controlado serão ligadas, desligadas ou desativadas a um dado momento. Usada para modularizar tarefas de controlo e peças separadas que podem ser executadas sequencialmente ou paralelamente, assim como controlar a sua execução. A norma IEC 61131-3 enfatiza a importância do *SFC* como uma "ajuda para estruturar programas de CLP".
LD	*Ladder Diagram*. Linguagem gráfica de ligação semelhante a conexões elétricas, de variáveis booleanas (relés e bobines), cuja visão geométrica se reflete como semelhante aos antigos controlos por relés. Os POU (ver capítulo 2.2.1) escritos em LD são separados em secções denominadas *networks*.
FBD	*Function Block Diagram*. Linguagem gráfica de ligação de elementos e blocos funcionais. Os POU escritos em *FBD* estão separados em secções denominadas *networks*. FBDs booleanos podem, normalmente, ser representados em LD e vice-versa.
IL	*Instruction List*: linguagem textual de baixo nível orientada à máquina, disponibilizada pela maioria dos sistemas de programação.
ST	*Structured Text*: linguagem textual de alto nível (semelhante ao *PASCAL*[3]) usada normalmente em tarefas de controlo e cálculos matemáticos complexos.

Refira-se que o presente trabalho foca-se no uso da linguagem IL. O modelo de *software* e programação é descrito através das suas definições formais, lexicais, sintáticas e semânticas. A norma é ainda complementada com exemplos do uso das linguagens.

2.2.1 ELEMENTOS CHAVE DA MODELAÇÃO EM IEC 61131-3

A evolução das linguagens proprietárias e normas locais para a norma IEC 61131-3 requereu uma mudança estrutural na modelação e programação, como no exemplo da Figura 2.5. A estrutura normalizada refere todos os blocos como alguma forma de POU (*Programmable Organizational Unit*).

[3] *PASCAL* – uma linguagem de programação publicada em 1970 que influenciou a história da programação, tendo uma abordagem estruturada e imperativa.

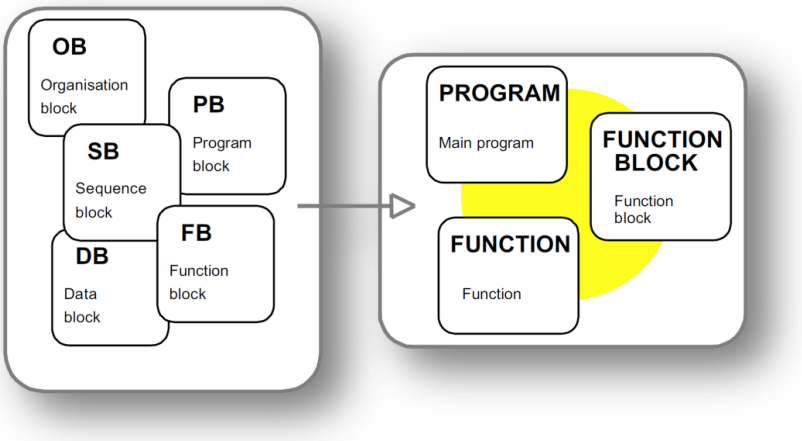

Figura 2.5 - A evolução dos antigos tipos de blocos do DIN 19239[4] para os POU do IEC 61131-3 [13]

Os POU são os blocos base da estrutura da programação de acordo com o IEC 61131-3, correspondendo aos variados tipos de blocos em sistemas de programação convencionais. Os POU podem chamar-se entre si, com ou sem passagem de parâmetros. São as mais pequenas unidades independentes de programa de controlo.

Existem 3 tipos de POU: POU função, POU bloco funcional e POU programa, em ordem crescente de funcionalidade, sendo a forma da sua chamada definida pelo IEC 61131-3. As diferenças entre eles estão descritas na Tabela 2.2. Uma listagem completa dos POU função e POU bloco funcional, que a norma IEC 61131-3 suporta, pode ser consultada nos Anexo I e II, respetivamente.

Tabela 2.2 - Os 3 tipos de POU e sua descrição

Tipo de POU	Nome simbólico	Características
Função	FUNCTION	O mais simples. Devolve um resultado diretamente previsível, pois não tem memória. Ex.: aritmética e comparações.
Bloco Funcional	FUNCTION_BLOCK	Tem registo de memória, podendo ser instanciado. Ex.: contadores e temporizadores.
Programa	PROGRAM	O programa de controlo. Tem acesso às entradas e saídas (I/O) do CLP, tornando-as acessíveis aos outros POU.

[4] Norma alemã de PLC, da *Deutsches Institut für Normung e. V.*

Os valores de memória utilizados na programação são os de variáveis, cuja declaração é textual, independentemente da linguagem de programação utilizada. A declaração e o uso são agnósticos relativamente ao CLP, sendo gerido pelo sistema de programação, não tendo os utilizadores que usar endereços de memória diretos, apesar de essa funcionalidade também ser suportada. As variáveis podem ser de vários tipos (booleanas, inteiros, etc.) assim como a sua natureza de I/O, podendo ser declaradas fora dos POU, para serem acessíveis a outros POU, serem passadas como parâmetros de um POU ou ainda com significado meramente local.

2.2.2 PROGRAMAÇÃO DE CLP ANTES E DEPOIS DO IEC 61131-3

Até à implementação da norma IEC 61131, só havia uma forma, economicamente viável, de ingressar num projeto de automação de grandes dimensões: escolhendo, de um fabricante, a opção proprietária mais adequada ao projeto e empresa [17]. Esta escolha de fabricante era aplicada a todo o projeto e mantida para o futuro. Este cenário proveio de um desenvolvimento disperso e proprietário do mercado dos CLP. Tipicamente, cada fabricante tem o seu equipamento único, com linguagens e ambientes de programação exclusivos. Suster equipas com as formações adequadas para lidar com múltiplos equipamentos diferentes entre si, conjugados com múltiplos ambientes de trabalho e linguagens de programação, todas diferentes com construção e sintaxe particulares, é reconhecidamente pouco frutífero num ambiente empresarial [18].

Com o advento da norma 61131-3, a indústria tem convergido para ambientes de programação que suportam todas as 5 linguagens de programação definidas. Com todas as grandes marcas a fornecerem suporte [19] com equipamentos e ambientes de desenvolvimento de programação que cumprem total ou parcialmente a norma, é hoje possível desenvolver e manter um projeto de automação com recurso a linguagens e equipamento em conformidade com a norma IEC 61131, inclusive em conformidade paralela com as normas de segurança IEC 61499 e IEC 61508 [20][21]. Já foi também referido que a norma 61131 se encontra incompleta [22][23], estando contemplada a criação de novos blocos funcionais específicos para os fabricantes/utilizadores [16].

2.2.3 IMPORTÂNCIA DO IEC 61131-3

As vantagens da implementação do IEC 61131 são várias. O IEC 61131 introduziu, de forma normalizada, novas ferramentas modernas de programação para os CLP, entre elas:

- Controladores PID.

- Lógica difusa.

Até então, estas ferramentas de controlo eram de uso limitado, proprietárias ou até inacessíveis [17]. Com a normalização da programação, pode escolher-se o conjunto *hardware-software* que mais se adequar ao empreendimento de automação, pois todos os modelos e marcas possuem vantagens e desvantagens [17]. Isto apenas é possível se todos os ambientes de programação e desenvolvimento seguirem a mesma norma de programação, sintaxe e funções suportadas.

A normalização também reduz ou elimina os custos de formação para equipas de programação/manutenção, podendo até reduzir o tamanho das equipas [24]. Com a escolha entre 5 linguagens de programação distintas, a modelação de engenharia da solução ao problema de automação é também beneficiada. Pode-se utilizar a linguagem mais adequada (ver Tabela 2.3) para o problema específico e encapsulá-la transparentemente.

Programas, funções e blocos funcionais tornam-se também reutilizáveis, reduzindo custos na criação de novos projetos e manutenção de antigos. Poderão ser ainda transplantáveis para outros equipamentos [25], com mais ou menos trabalho de adaptação (exemplo: converter blocos funcionais associados especificamente a um equipamento, para outro), de acordo com a compatibilidade e conformidade com a norma 61131-3. Isto já foi feito com elevado sucesso [18].

Tabela 2.3 - Aplicações típicas das várias linguagens do IEC 61131-3 [9]

Linguagem	Linguagem/Funcionalidade	Aplicações típicas
LD	Diagrama de circuitos	Circuitos de controlo elétrico, como interruptores e lâmpadas.
IL	Linguagem do tipo *assembly*[5]	Módulos de controlo com comportamento temporal crítico.
FBD	Operações e funções booleanas	Comunicação, funções reutilizáveis, controlo e integração.
SFC	Diagrama de estados	Sequenciação de controlo.
ST	Linguagem de alto nível	Funções tecnológicas, controlo de alto nível.

[5] *Assembly* é uma linguagem de programação de baixo nível, logo acima de código-máquina. É composta por mnemónicas que correspondem diretamente a código-máquina de execução.

A indústria da automação recebeu de bom grado [19] as vantagens que advêm do IEC 61131-3, reduzindo custos e tempos de espera e transição, melhorando eficácia e melhorando a qualidade dos produtos.

Dado o suporte dos fabricantes à norma 61131 (ver 2.2.6), surgiram novos ambientes de programação que cumprem a norma e são compatíveis com os equipamentos modernos. Estes possibilitam aproximações de programação modernas, como a modelação, *debugging* e deteção de erros.

Todas estas características tornam possível a colaboração de equipas com formações e capacidades diferentes. Isto reflete-se também na educação e aprendizagem da programação de CLP [26], principalmente com novos programadores: estes não têm que escolher entre diversas linguagens e plataformas de desenvolvimento específicas a cada fabricante [27], tornando a aprendizagem das linguagens do IEC 61131-3 propedêutica e aplicável em qualquer meio, com pouca ou nenhuma formação extra [24].

2.2.4 EXPANDINDO A NORMA – O *PLCOPEN*

Formada em 1992, imediatamente após a publicação da norma IEC 61131-3, o *PLCopen* [28] é uma organização não lucrativa cujo objetivo é disseminar, apoiar e certificar quanto ao cumprimento da norma. A gestão técnica do *PLCopen* está organizada em comissões técnicas com diferentes papéis (Figura 2.6).

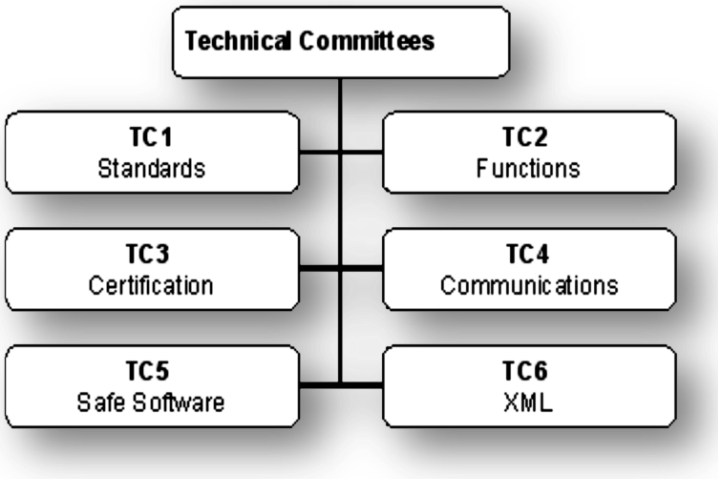

Figura 2.6 – As diferentes comissões técnicas do *PLCopen* [16]

Não tendo nenhum fabricante ou produtos afiliados, o *PLCopen* foca-se na harmonização da programação dos CLP e desenvolvimento de *software* e aplicações no ambiente IEC 61131-3, como bibliotecas comuns e níveis de conformidade com a norma. Existem vários níveis de conformidade com a norma 61131-3 e vários níveis de certificações, atribuídas pelo *PLCopen* [28]. Estas certificações são dadas pelo comité TC3 do *PLCopen* e são testadas por uma organização acreditada pela mesma. As certificações são as descritas na Tabela 2.4.

Tabela 2.4 - Certificações de cumprimento da norma IEC 61131-3 de acordo com o *PLCopen*

Nível de certificação	Descrição da conformidade
Base Level (BL)	O "nível base". Indica conformidade com sintaxe correta, programação e metodologia da estrutura básica de um programa.
Reusability Level (RL)	O "nível de reutilizabilidade". As funções e blocos funcionais são compatíveis e transplantáveis para outros equipamentos com a certificação RL.
Conformity Level (CL)	O "nível de conformidade". É o nível mais elevado de conformidade, quanto ao funcionamento e estrutura de um programa.

Esta organização definiu ainda um formato de exportação de projetos, programas, funções ou blocos funcionais escritos de acordo com a norma, de um ambiente de programação para outro.

2.2.5 IMPORTAÇÃO/EXPORTAÇÃO EM XML

Apesar de prever portabilidade de código, a norma IEC 61131-3 não define nenhum formato para a exportação/importação de projetos entre ambientes de desenvolvimento. O seu suporte limita-se a blocos de texto [13], onde a sua utilidade reflete-se mais na comunicação com os CLP do que propriamente com desenvolvimento de projetos. Pela necessidade de tal formato normalizado, o *PLCopen* propôs, em 2005, um *schema* XML (*eXtended Markup Language* [29]) que foi rapidamente adotado e implementado pela indústria [28]. É gerido pela TC6 do *PLCopen* e o *schema* XML de modelo de dados e validação é conhecido como TC6-XML. Um *schema* consiste numa descrição de um tipo de documento XML, que inclui os termos de validação de tais documentos, indicando características tais como:

- Estrutura do documento.
- Tipos de conteúdo.
- Imposição sintáticas.
- Imposições gramáticas.

Através destas características, é possível validar e apenas aceitar documentos em XML que cumpram os requisitos funcionais para o sistema computacional funcionar adequadamente. O formato de portabilidade permite uma representação completa de um projeto de acordo com a norma IEC 61131-3, incluindo: linguagens de programação textual (IL e ST), linguagens gráficas (LD e FBD), linguagem estrutural (SFC), informação gráfica, ligações de portos, comentários, os POU, tipos de variáveis, informação de projeto, etc. Ou seja, contém tudo de um projeto, mas num formato aberto e transmissível entre ambientes de desenvolvimento.

2.2.6 IMPLEMENTAÇÃO NO MUNDO DA AUTOMAÇÃO INDUSTRIAL

Atualmente é essencial referir o IEC 61131 em qualquer trabalho ou estudo relacionado com CLP. As grandes marcas de fabricantes de CLP (como algumas das ilustradas na Figura 2.7) têm aderido com suporte de *hardware* e *software* a corresponder, mas também fornecendo comentários e trabalhando junto com o IEC e o *PLCopen* para uma norma mais adequada às necessidades do mercado [18].

Figura 2.7 – Algumas organizações envolvidas no desenvolvimento do IEC 61131

O suporte de *software* é dado através dos ambientes de desenvolvimento, os IDE, dos fabricantes de CLP, sendo já vários os que suportam a norma do IEC. Juntamente com o suporte de *hardware*, a implementação prática da norma começa a ter uma grande força de mercado, tendo os grandes fabricantes já apostado no suporte da norma IEC 61131-3 nos seus IDEs, tais como:

- A *Siemens* com o *Step 7* [30].
- A *Schneider Electric* com o *Concept* [31].

Outras marcas enveredaram por outro caminho, tendo licenciado versões próprias de IDEs mais abrangentes, tais como:

- *MultiPROG* da *KW Software* [32] que suporta vários fabricantes (*ABB*, *Advantech*, *Baumüller*, *Bristol Babcock*, *Fuji*, *Hitachi*, *Kuka Roboter*, *Mitsubishi*, *MAN Roland*, *Phoenix Contact*, *Schleicher* e *Yokogawa*).
- *CoDeSys* da *S3* [33] suportando o *IndraLogic* da *Bosch Rexroth* [34], *TwinCAT* da *STA* [35], *X Soft* da *Klockner Möeller* [36] e o *I/O Pro* da *Wago* [37].

Estes ambientes suportam o IEC 61131-3 na totalidade, mas com níveis variáveis de cumprimento da norma (ver capítulo 2.2.4), para cada linguagem, sendo possível utilizar apenas comandos e instruções que sigam plenamente a norma. Existem ainda alternativas gratuitas para editar projetos, como:

- *PLCOpen Editor* [38] (Figura 2.8).
- *OpenPCS Automation Suite* [39].

Estes editores permitem a importação/exportação em TC6-XML, sendo capazes de editar várias propriedades de projeto (organização e máquina-destino), elementos independentes (tipos definidos pelo utilizador), declaração de variáveis, edição de propriedades das variáveis e editar todos os programas, funções e blocos de funções em qualquer das linguagens. O *PLCOpen Editor* (Figura 2.8) tem a particularidade de estar disponível em código aberto[6], enquanto que o *OpenPCS* oferece funcionalidades de simulação de CLP.

São várias as ferramentas de auxílio à programação oferecidas por estes IDEs modernos, onde cada um terá o seu conjunto próprio, onde podem incluir-se:

- Conversões automáticas entre linguagens do IEC 61131-3 – como, por exemplo, converter código em LD para IL ou ST, apesar destas conversões estarem limitadas por pequenos erros de sintaxe, códigos exóticos, funções não suportadas, etc...
- Suporte à portabilidade de código através do TC6-XML (ver capítulo 2.2.4) – códigos em IL (e ST) podem ser diretamente importados/exportados por texto, havendo normalmente apenas a necessidade de acertos quanto a funções não suportadas ou pequenas variações de sintaxe.

[6] Código aberto, também conhecido como *open source*, define software com distribuição livre, acesso ao código fonte, permite trabalhos derivados mas mantendo a integridade do código do autor.

- Ferramentas de *debug* modernas – como remate automático de palavras-chave, *breakpoints*[7] e análise de valores internos de variáveis.
- Simulação do funcionamento do CLP – permite prever o comportamento do CLP com o programa em desenvolvimento.

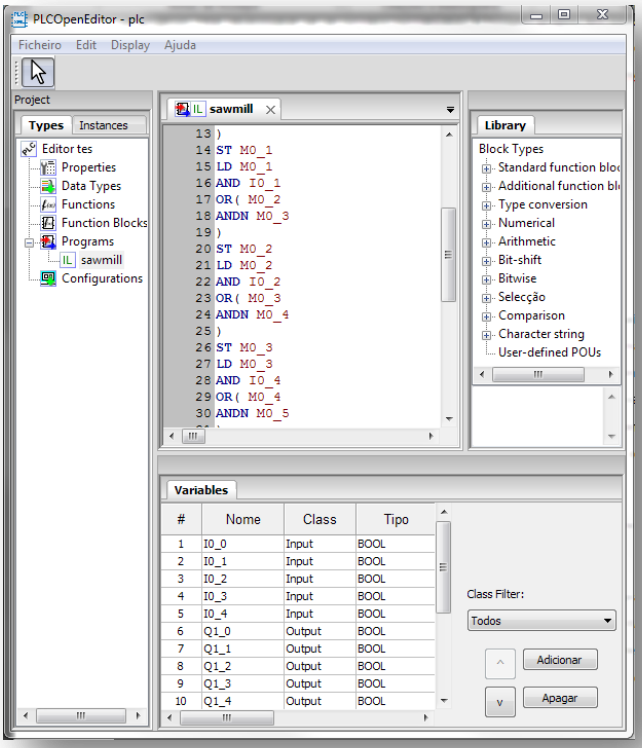

Figura 2.8 - O *PLCOpen Editor* [38]

[7] Pausas de processamento em linhas de código predeterminadas, com seguimento passo a passo.

3 SIMULAÇÃO DE PROCESSOS

A implementação de uma linha de produção industrial envolve, normalmente, um grande investimento, onde cada decisão de projeto é crucial para assegurar o funcionamento pretendido. Uma linha de produção industrial moderna é um sistema altamente integrado composto por vários autómatos, cada um com múltiplas ferramentas e processos, controlados por CLP, e um sistema controlado por computador de gestão do processo completo. Uma questão chave é a programação dos CLP, onde um pequeno erro pode comprometer toda a linha de produção. É por isso que é essencial testar, validar e analisar programas de controlo de CLP. É aqui que entra a modelação e simulação.

3.1 MODELAÇÃO E SIMULAÇÃO DE PROCESSOS INDUSTRIAIS

A simulação de processos oferece muitas vantagens, principalmente no contexto de testes de controlo de processos industriais complexos, tais como [9]:

- Múltiplas simulações encadeadas e/ou paralelas – vários programadores a executarem testes ou várias configurações de teste.
- Simulação em tempo diferido – possibilita uma análise temporal com escrutínio impossível no processo real, ou uma análise mais rápida para análise de resultados finais.
- Análises minuciosas, que seriam impossíveis num teste real.
- Custos reduzidos de teste drasticamente reduzidos.
- Risco de danos na maquinaria eliminado – uma mais-valia para qualquer empreendimento!

Para a simulação ser possível, é necessário modelar o processo em estudo, através de um modelo matemático que o descreva. Este modelo pode ser utilizado em ferramentas de simulação por computador (como ambiente *Matlab*/*Simulink*, descrito no capítulo 5.1.2) ou para fins de análise e projeção dos sistemas de controlo. A modelação de um processo industrial pode ser realizada através de uma função de transferência para sistemas lineares, em forma em contínua ou discreta, com um conjunto de entradas e saídas. As entradas do modelo correspondem aos atuadores do processo e as saídas correspondem a valores obtidos através de uma rede de sensores e detetores.

A modelação acarreta várias vantagens em termos de testes e simulação, mas também apresenta desvantagens, que advêm de uma modelação imperfeita: um comportamento divergente relativamente ao processo real. Apesar de tudo, estes desvios podem ser detetados e retificados.

Para além do processo em estudo, ainda falta modelar o próprio controlo (ver capítulo 1.2) para se poder completar a simulação.

3.2 SOLUÇÕES ATUAIS

Como já foi referido, existe uma lacuna no binómio desenvolvimento/testes de programa [6][10]. As abordagens clássicas (modelos à escala e baterias de interruptores/indicadores luminosos) revelam-se cada vez mais inadequadas, apenas aptas para processos simplistas. Para processos complexos, existem várias abordagens de simulação e teste de CLP computadorizadas, tais como [2][3][4]:

- Sistemas SCAD e IHM.
- Ferramentas de simulação baseados em computador, como o *PC-SIM* [7] ou o *PSIM* [8].

Recentemente (2010), surgiu outra alternativa prática à solução proposta, com algumas semelhanças de abordagem: o pacote de conversão *Simulink PLC Coder* [40] (Figura 3.1).

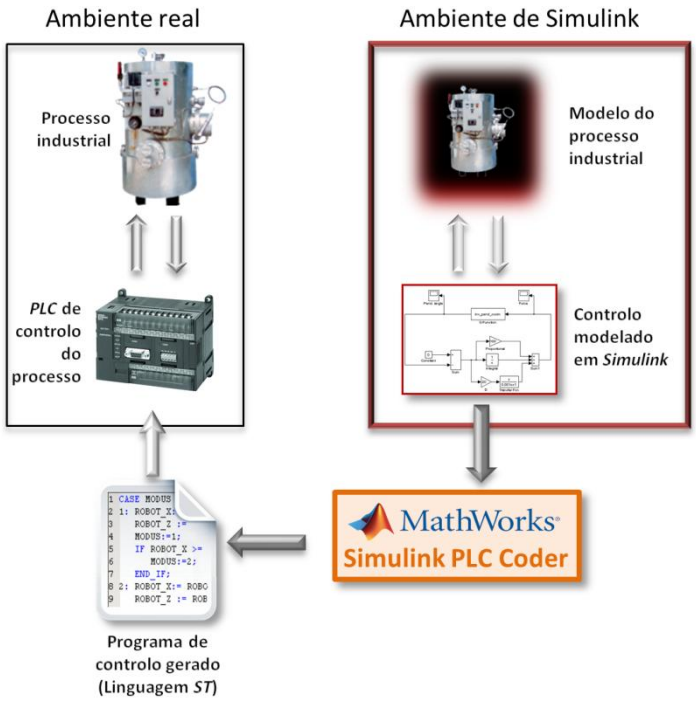

Figura 3.1 - Visão geral do uso do *Simulink PLC Coder*

Como ilustrado na Figura 3.1, o processo é inverso à solução proposta (ilustrada na Figura 1.3). Em vez de se gerar o modelo equivalente para simulação, gera-se o programa de CLP a partir do controlo modelado no ambiente de simulação. Especificamente e no caso do ambiente de simulação *Matlab*/*Simulink*, isto é possível com recurso ao *Simulink PLC Coder*. Este, gera código em ST (ver Tabela 2.1) de acordo com o IEC 61131-3, a partir de modelos *Simulink*, estadogramas *Stateflow* e funções embutidas do *Matlab*. Este código é exportado em formato TC6-XML (capítulo 15) direcionado para um IDE/fabricante específico. A Figura 3.1 ilustra o seu funcionamento geral. É um método válido para atacar o problema exposto neste trabalho, mas que apresenta algumas desvantagens face à solução proposta. O contraste entre as diferentes abordagens possíveis pode ser consultado na Tabela 3.1, onde são analisadas qualitativamente as suas vantagens e desvantagens.

Como é possível verificar, a solução proposta apresenta vantagens em todos os campos. Apenas o recurso *Simulink PLC Coder* apresenta vantagens semelhantes, mas contendo ainda alguns impedimentos graves, especialmente no que toca à formação extra necessária e educação de programação de CLP.

Tabela 3.1 – Qualificações em vários campos das abordagens ao problema da simulação, face à solução proposta

Abordagem / Campo	Sistemas IHM e SCAD	Ferramentas de simulação por computador	*Simulink PLC Coder*	Solução proposta
Educação da programação de CLP	Suficiente. Muito limitada pelo suporte de processos e CLP.	Suficiente. Muito limitada pelo suporte de processos e CLP.	Inútil. Toda a modelação do controlo passa a ser feita no ambiente de simulação.	Vantajosa, permitindo efetuar testes em ambiente simulando, utilizando programação documentada.
Desenvolvimento de programas de controlo	Suficiente. Muito limitada pelo suporte de processos e CLP.	Suficiente. Muito limitada pelo suporte de processos e CLP.	Adequada. Apenas peca pela imprevisibilidade aquando da execução no ambiente real.	Vantajosa. A experiência prévia de outros programas permite uma abordagem mais segura.
Abrangência de processos	Muito limitada. Apenas processos suportados pelo fabricante.	Muito limitada. Apenas processos suportados pelo pacote de simulação.	Vantajosa. Praticamente qualquer processo pode ser modelado.	Vantajosa. Praticamente qualquer processo pode ser modelado.
Abrangência de CLP	Muito limitada. Associada a protocolos proprietários.	Suficiente. Suporte limitado a algumas linguagens.	Suficiente. Limitada a CLP modernos que suportem a norma.	Vantajosa. Grande abrangência, expansível a várias linguagens proprietárias.
Formação extra necessária	Formação do uso das ferramentas, que incluirá a integração de modelos e programas de controlo.	Pouca ou nenhuma. Retém-se o uso do IDE escolhido e apenas se requere formação do uso do ambiente de simulação.	Necessária formação completa da modelação de controladores no ambiente de simulação.	Pouca ou nenhuma. Retém-se o uso do IDE escolhido e apenas se requere formação do uso do ambiente de simulação.
Custo extra	Considerável.	Nenhum.	Licenças do pacote de conversão.	Nenhum.

4 SOLUÇÃO CONCEPTUAL

A solução conceptual será apresentada através de 3 visões: global, funcional e arquitetural. A visão global mostrará toda a abrangência do trabalho, enquanto que as visões funcional e arquitetural serão específicas a cada um dos módulos desenvolvido no presente trabalho.

As diferentes visões são apresentadas recorrendo à norma do desenvolvimento de sistemas computacionais, que é o UML (*Unified Modeling Language*) [41]. O UML é uma linguagem de modelação que inclui vários tipos de notação gráfica, entre os quais estão os utilizados neste trabalho:

- Diagrama de casos de uso (*Use Case Diagram*). Representação comportamental de todas as ações que são disponibilizadas pelo sistema ao utilizador (ator).
- Diagrama de sequência (*Sequence Diagram*). Representação comportamental de como a diferentes entidades interagem entre elas, porque que ordem e com que elementos.
- Diagrama de atividades (*Activity Diagrams*). Representação comportamental de procedimentos passo a passo de atividades.
- Diagrama de classe (*Class Diagram*). Representação estrutural das classes do sistema, incluindo atributos e métodos.

4.1 VISÃO GLOBAL

Este trabalho pressupõe a existência de um modelo de processo em estudo, ou a possibilidade da sua modelação. Este modelo de processo estará, tipicamente, associado a um dado ambiente de simulação onde se pretende replicar também o programa de controlo original (como ilustrado na Figura 1.3), através de um processo de conversão, foco do trabalho desenvolvido. O resultado desta conversão é aceite pelo ambiente de simulação, sendo específico ao programa originalmente em código CLP e tendo um comportamento equivalente. Será gerado a partir do programa original, cujo percurso completo, desde o programa de controlo em código proprietário, até ao modelo equivalente para integração no ambiente de simulação está ilustrado na Figura 4.1.

Figura 4.1 – Percurso completo da geração do modelo equivalente a partir de código proprietário

Para os casos de código original em formato proprietário (*i.e.* outra linguagem de programação de CLP não de acordo com as normas internacionais), o código tem primeiro que ser "normalizado", para um formato mais uniformizado. Este é um passo crucial pois existem quase várias linguagens de programação proprietárias de CLP. Para isso, são utilizados "dicionários de regras de tradução", únicos para cada linguagem proprietária. Com o código de controlo já conforme as normas internacionais abrangentes, aplicar-se-á o processo de "conversão" para ambiente de simulação.

Torna-se então possível, através do trabalho desenvolvido, simular qualquer programa de controlo CLP, mesmo que escrito numa linguagem proprietária, desde que um dicionário de regras de tradução adequado seja criado. Os pacotes "Conversor com recurso a normas internacionais" e "Tradutor com recurso a normas internacionais", alvos de desenvolvimento no trabalho apresentado, serão referidos daqui em diante como, **Matlaber** e **UnifIL**, respetivamente.

4.2 VISÃO FUNCIONAL

A visão funcional demonstra as funcionalidades disponibilizadas aos utilizadores em cada sistema, sendo únicas para cada ferramenta desenvolvida.

4.2.1 VISÃO FUNCIONAL PARA *MATLABER*

O diagrama de casos de uso **Matlaber**, que pode ser consultado na Figura 4.2, ilustra as funcionalidades disponibilizadas, que são estritamente as necessárias. O ator corresponde ao utilizador, que ativa todo o processo.

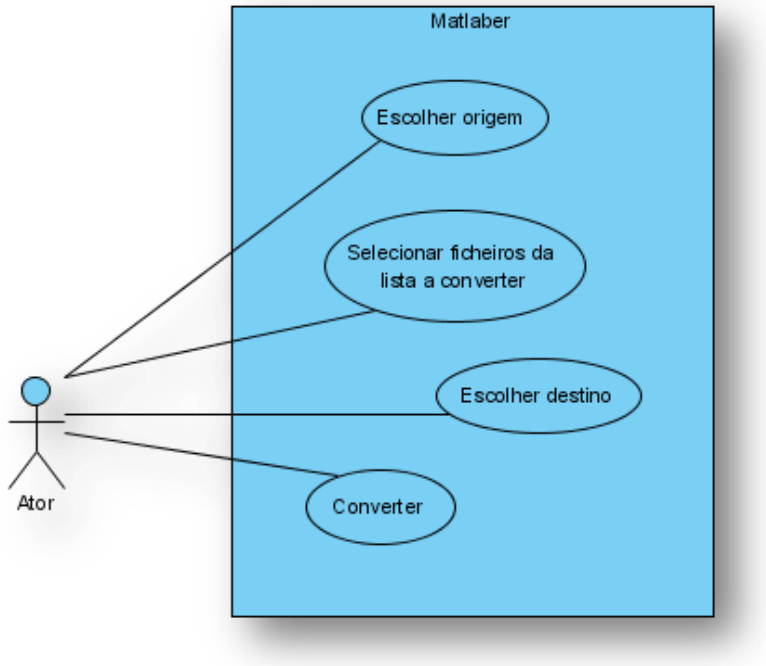

Figura 4.2 – Diagrama de casos de uso do *Matlaber*

As funcionalidades disponibilizadas são:

- **Escolher origem** – escolha da origem dos ficheiros (localização ou diretoria) para que o interface possa listar os ficheiros disponíveis para conversão.
- **Escolher ficheiros da lista a converter** – escolha dos ficheiros a converter, da lista atual.
- **Escolher destino** – escolha do destino dos ficheiros (localização ou diretoria) para onde o conversor gerará os ficheiros convertidos.
- **Converter** – ativar a conversão dos ficheiros selecionados.

4.2.2 VISÃO FUNCIONAL PARA *UNIFIL*

O diagrama de casos de uso **UnifIL**, que pode ser consultado na Figura 4.3, ilustra as funcionalidades disponibilizadas, que são estritamente as necessárias. O ator corresponde ao utilizador, que ativa todo o processo.

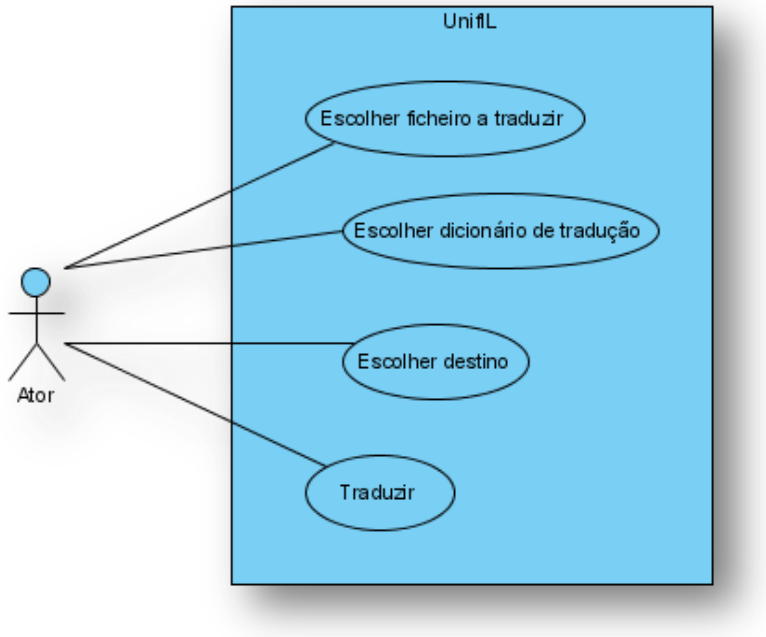

Figura 4.3 - Diagrama de casos de uso do *UnifIL*

As funcionalidades disponibilizadas são:

- **Escolher ficheiro a traduzir** – escolha do ficheiro a traduzir.
- **Escolher dicionário de tradução** – escolha do dicionário a utilizar na tradução.
- **Escolher destino** – escolha do destino dos ficheiros (localização ou diretoria) para onde o conversor gerará os ficheiros convertidos.
- **Traduzir** – ativar a tradução do ficheiro com o dicionário escolhido.

4.3 VISÃO ARQUITETURAL

A arquitetura do modelo conceptual é constituída pelas 3 camadas (Interface, Controlo e Entidade) da arquitetura ICE (*Interface*, *Control* e *Entity*), cujo propósito é o de guiar o desenvolvimento de sistemas computacionais. Esta é específica para cada um dos sistemas computacionais desenvolvidos e será explicada para cada um.

4.3.1 VISÃO ARQUITETURAL DO MATLABER

A arquitetura conceptual do **Matlaber** consiste nas 3 camadas ilustradas na Figura 4.4.

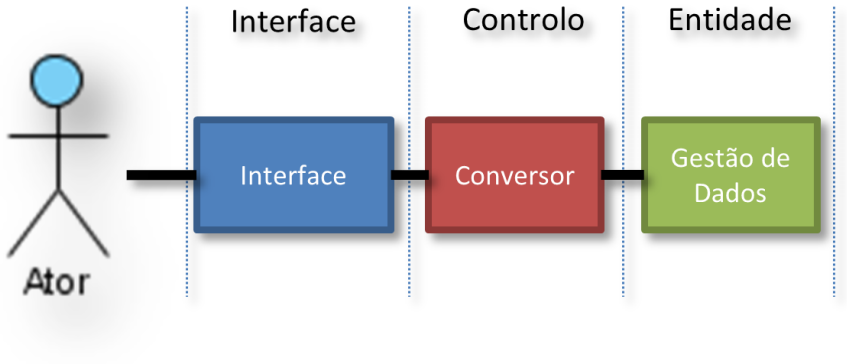

Figura 4.4 – Arquitetura ICE conceptual do *Matlaber*

Cada camada possui uma classe, com a ligação estrutural entre classes relacionadas todas entre si, como ilustrado na Figura 4.5.

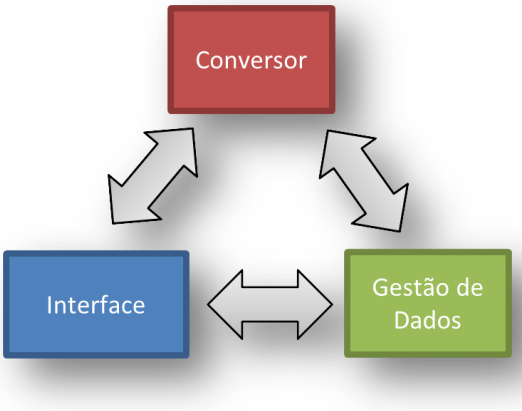

Figura 4.5 - Relações entre as diferentes classes do *Matlaber*

Segue-se a descrição das classes e as suas ligações às restantes:

- **Classe *Interface***: esta classe é responsável por permitir ao utilizador usufruir as funcionalidades indicadas no diagrama de casos de uso do **Matlaber** (Figura 4.2), necessitando para isso de acesso à classe *Gestão de Dados* para poder indicar os dados relevantes e chamando a classe *Conversor*.

- **Classe *Conversor***: classe que efetua a conversão dos ficheiros indicados pela classe *Interface*, através de um processamento dinâmico de conversão do código. Faz também recurso a códigos previamente implementados dos POU através da *Gestão de Dados*.
- **Classe *Gestão de Dados*:** esta classe disponibiliza o acesso à leitura e escrita de ficheiros, tanto para a classe *Conversor* como para a classe *Interface*, para além de disponibilizar os códigos dos POU.

4.3.2 VISÃO ARQUITETURAL DO UNIFIL

A arquitetura conceptual do **UnifIL** consiste nas 3 camadas ilustradas na Figura 4.6.

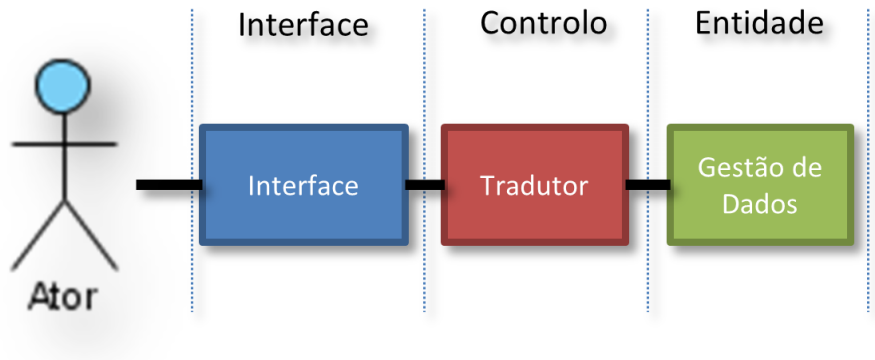

Figura 4.6 - Arquitetura ICE conceptual do *UnifIL*

Cada camada possui uma classe, com a ligação estrutural entre classes relacionadas todas entre si, como ilustrado na Figura 4.7.

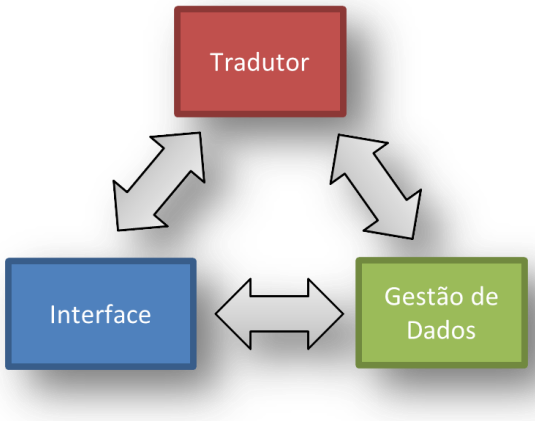

Figura 4.7 - Relações entre as diferentes classes do UnifIL

Segue-se a descrição das classes e as suas ligações às restantes:

- **Classe *Interface***: esta classe é responsável por permitir ao utilizador usufruir as funcionalidades indicadas no diagrama de casos de uso do **Unifil** (Figura 4.3), necessitando para isso de acesso à classe *Gestão de Dados* para poder indicar os dados relevantes e chamando a classe *Tradutor*.
- **Classe *Tradutor***: classe que efetua a tradução dos ficheiros indicados pela classe *Interface*, através de um processamento dinâmico de conversão do código, com recurso a dicionários disponíveis através da classe *Gestão de Dados*.
- **Classe *Gestão de Dados***: esta classe disponibiliza o acesso à leitura e escrita de ficheiros, tanto para a classe *Tradutor* como para a classe *Interface*, para além de disponibilizar os dicionários de tradução.

5 IMPLEMENTAÇÃO

Este capítulo descreve a implementação do trabalho, desenvolvido para suportar a solução conceptual previamente apresentada, nomeadamente, os 2 sistemas computacionais desenvolvidos: **Matlaber** e **UnifIL**.

5.1 DECISÕES DE DESENVOLVIMENTO E IMPLEMENTAÇÃO

Durante o desenvolvimento do trabalho apresentado foram tomadas decisões importantes no desenvolvimento, que têm que ser justificadas. Foram elas:

- Recurso ao IEC 61131-3 e a linguagem IL como linguagem alvo.
- *Matlab/Simulink* como ambiente de simulação alvo.
- Integração e representação em *Simulink*.
- Conversão de natureza simbólica.
- Integração do TC6-XML do *PLCopen*.
- Uso de dicionários de regras de tradução em XML.

Serão justificadas e explicadas detalhadamente nas secções abaixo.

5.1.1 RECURSO AO IEC 61131-3 E A LINGUAGEM IL

A norma IEC 61131-3 tem tido um suporte crescente, sendo hoje apoiada pela maioria dos fabricantes, com ênfase para os fabricantes-chave como a *Siemens* e *Schneider* [30][31]. Está-se a tornar no modelo *de facto* para o desenvolvimento de *hardware* e *software* CLP. Para o trabalho desenvolvido ter alguma relevância, terá que ser orientado à norma IEC 61131-3.

Todas as linguagens do IEC 61131-3 (descritas na Tabela 2.1) estão bem documentadas, mas a linguagem de programação mais utilizada na programação de CLP é o LD [42], seguida pelo IL. É uma linguagem gráfica que não corre nativamente nos CLP, sendo convertida para uma linguagem intermediária textual: o IL, semelhante a *assembly*. O LD é, portanto, diretamente convertível para IL, geralmente através do próprio IDE de programação mas também através de outros processos externos [42][43]. Esta capacidade de conversão torna o IL tão abrangente quando o LD, com uma complexidade inerente reduzida por ser uma linguagem textual. Possui

um reduzido número de instruções (ver capítulo 5.3.3.3), cuja chamada é simples e pouco variável, o que impede uma escalada exponencial da complexidade de implementação.

Estas características tornam a linguagem IL mais acessível para um suporte mais completo por parte do trabalho desenvolvido, *i.e.* o maior alcance de instruções reconhecidas e serem correta e fielmente convertidas para o ambiente de simulação, mantendo o trabalho passível de ser executado no âmbito da dissertação de mestrado e tempo disponível. Torna-se então claro o porquê da escolha da linguagem IL como alvo do trabalho apresentado.

5.1.2 O *MATLAB* COMO AMBIENTE DE SIMULAÇÃO ALVO.

O *Matlab* é uma referência em simulação [44], tanto no meio académico como na indústria, desde análise económica até simulações de órbitas de satélites. É um pacote de *software* extremamente completo, sendo ainda bastante flexível e aberto, permitindo a criação de módulos, funções, blocos de simulação, etc. Os modelos de processos já existentes para *Matlab*/*Simulink* são inúmeros, sendo várias as ferramentas disponíveis para os criar, tendo ainda uma vasta documentação disponível. Também possibilita a integração de outras ferramentas de modelação. Isto aumenta o alcance e flexibilidade do *Matlab*, pois cada pacote de simulação disponível é superior numa dada área de investigação, como: *Modelica* (ferramenta de modelação física) [45], *SPICE* (modelação de eletrónica) [46] ou o *ASPEN Plus* (modelação de processos químicos) [47].

Apesar não ter uma tão grande expressão especificamente na indústria da automação [48], a sua flexibilidade e capacidade de integrar outras ferramentas de modelação tornam o *Matlab* bastante versátil e o ambiente de simulação ideal para a solução proposta.

O ambiente *Simulink* é uma plataforma de projeção gráfica baseada em modelos e blocos, estando completamente integrada no *Matlab*. As bibliotecas de blocos e modelos são diversos e extensivos, abrangendo áreas como controlo (Figura 5.1), matemática, cálculo financeiro, modelação 3D, sistemas em tempo real, etc.

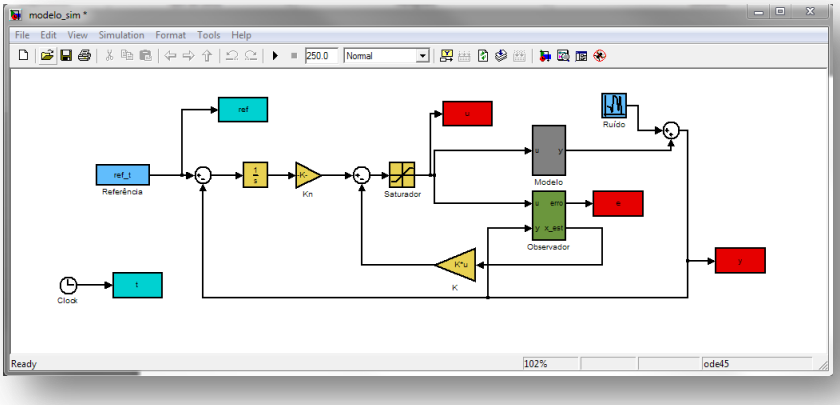

Figura 5.1 - Uma aplicação de controlo em ambiente *Simulink*

5.1.3 INTEGRAÇÃO E REPRESENTAÇÃO EM *SIMULINK*

A integração do trabalho apresentado com o *Matlab/Simulink* é feita através do **Matlaber**. Para simular completamente o programa de controlo do CLP, é necessário integrar o código convertido num bloco de controlo do *Simulink* – mais precisamente, num bloco "*MATLAB Function*", onde apenas é necessário mapear as entradas e saídas virtuais do CLP para outros portos virtuais. A Figura 5.2 é um exemplo de um esquema generalista *Simulink* para integração do trabalho.

O modelo de CLP equivalente (**a**) está integrado sob um bloco de função *Matlab* ("*MATLAB Function*"), com as devidas ligações de entrada (**b**) e saída (**c**), ambas escaláveis, para o modelo do processo (**d**), que possui um número definido de entradas e saídas. Existem entradas adicionais (**e**) que correspondem ao interface de comando, sinal de relógio (**h**), assim como saídas para além das ações de atuadores, como medidas (**f**) e gráficos (**g**). Este é um esquema generalista que terá sempre de ser adaptado tanto ao bloco do processo como ao bloco do CLP, até porque ambos poderão ter diversos tipos e números de entradas/saídas (analógicas e digitais) que terão de ser coerentes – tal como no processo real, é preciso ligar as saídas certas às entradas certas!

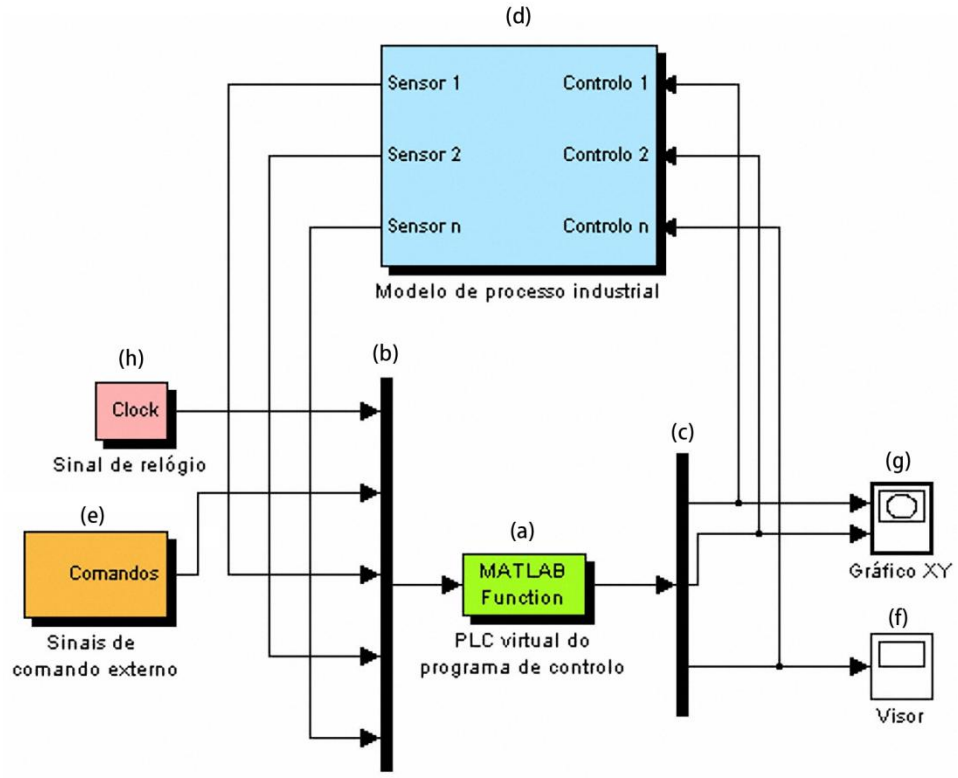

Figura 5.2 - Esquema generalista para integração no *Matlab*/*Simulink*, de acordo com a solução proposta

A função de controlo, sob a forma de texto num ficheiro ".m", que a "*MATLAB Function*" (**a**) chama recursivamente, deverá estar num formato coerente com as múltiplas entradas e saídas, independentemente do programa de controlo que execute.

5.1.4 CONVERSÃO DE NATUREZA SIMBÓLICA

O funcionamento clássico de uma linguagem de programação dos CLP semelhante ao IL (como o STL da *Siemens*) inclui o recurso a um acumulador[8] com várias camadas, em forma de pilha (*stack*). Este funcionamento é uma herança das limitações dos primeiros CLP, tendo sido completamente substituído com o advento da norma IEC 61131-3, através do suporte ao uso de isolamento de operações no código IL – *i.e.* escrita com recurso a parêntesis. É também uma forma de escrita de programação de difícil escrita/leitura para humanos, podendo ocupar várias linhas de código e recorrendo a várias operações da pilha de acumuladores, com algo que poderia ser reescrito muito mais clara e sucintamente com recurso ao isolamento de operações por parêntesis. A Figura 5.3 ilustra um exemplo como o descrito.

[8] Acumulador universal, valor em memória do resultado da última operação.

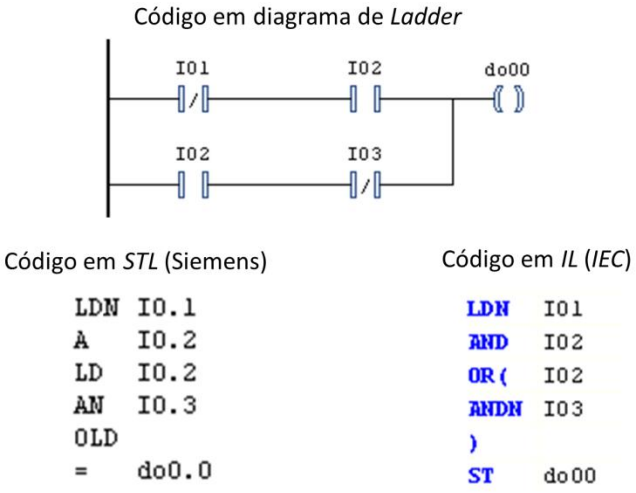

Figura 5.3 - Exemplo de código com recurso a pilha de acumuladores

A linguagem IL do IEC 61131-3 não inclui suporte a operações de pilha de acumulador, pelo que o trabalho desenvolvido terá que refletir 2 aspetos:

- Conversão adequada de operações com o acumulador do IL (desenvolvido no capítulo 5.3.3.3).
- Tradução adequada de operações de pilha de acumuladores, inerente a muitas linguagens de CLP proprietárias, para o acumulador único do IL. (desenvolvido no capítulo 5.4.3)

5.1.5 INTEGRAÇÃO DO TC6-XML DO *PLCOPEN*

O formato aberto de importação/exportação de projetos baseados no IEC 61131-3, o TC6-XML, foi muito considerado para a integração no trabalho apresentado. Mas foi ultimamente rejeitada, tanto na exportação como na importação. Segue-se a motivação para tal decisão:

- Exportação de TC6-XML – aplicável no processo de tradução de códigos proprietários (**UnifIL**), poderia ser útil para que os programas traduzidos fossem diretamente importados para algum IDE com suporte ao IEC 61131-3. Esta ideia foi rejeitada pois todas as restantes informações de projeto estariam vazias ou nulas, acabando por tornar o ficheiro de exportação um ficheiro complexo em XML com imensas informações em branco, só para transmitir um troço de código em texto simples. Esta abordagem seria contraproducente, pois as restantes informações iriam destabilizar as definições de projeto no IDE.

- Importação de TC6-XML – aplicável no processo de conversão para ambiente de simulação (**Matlaber**), poderia ser útil para importação de código de qualquer IDE com suporte ao IEC 61131-3. Apesar de, em princípio, não haver nenhum impedimento à implementação desta importação, existem ferramentas (ver capítulo 2.2.5) largamente disponíveis que farão sempre um melhor trabalho em extrair a porção de código relevante do ficheiro XML.

Tanto para a importação, como para a exportação, existem ferramentas adequadas (ver capítulo 2.2.6) para lidar com projetos em TC6-XML que fazem um melhor trabalho a gerar um ficheiro XML com informações de projeto e dados que estariam vazios de outro modo, assim como a leitura das várias informações de um projeto importado, tendo uma separação clara de todos os ficheiros de programa, de onde é fácil extrair o código relevante para conversão.

5.1.6 USO DE DICIONÁRIOS DE TRADUÇÃO EM XML

Para o trabalho desenvolvido, foi decidida a utilização de módulos separados de regras de tradução, os "dicionários" para uso no **UnifIL** (ver capítulo 5.1.6). Ao separar o tradutor das regras de tradução, obtém-se uma abordagem expansível a várias linguagens de programação de CLP proprietárias, ao não limitar o tradutor às linguagens suportadas de origem. São escritos em XML, contendo todos os elementos necessários à tradução do código proprietário. O XML é um conjunto de regras semânticas normalizadas, que permitem uma linguagem de marcação de dados flexível e legível por máquinas e humanos. Prevê ainda métodos de validação, que aumentam a robustez de qualquer sistema que o use, nomeadamente os *schemas*. Estes são também escritos em XML e compõem regras de validação para documentos em XML.

5.2 FERRAMENTAS UTILIZADAS

A implementação recorreu a várias tecnologias, descritas na Tabela 5.1, onde se inclui uma motivação para o seu uso.

Tabela 5.1 - Tecnologias utilizadas

Tecnologia	Descrição	Motivação
Visual Studio 2010	É um IDE disponibilizado pela *Microsoft* para desenvolvimento de *software* através de várias linguagens de programação, como C++, C#, VB.NET e J# [49].	Foi utilizado devido à sua modernidade (*debugger* muito completo, IDE com ajudas e guias de escrita inteligentes incorporados), familiaridade (é uma ferramenta comum para estudantes) e acessibilidade (disponível gratuitamente para os alunos).
C#	C# é uma linguagem de programação orientada a objetos desenvolvida pela Microsoft como parte da plataforma .NET.	A linguagem de eleição para o IDE escolhido, aplicando-se as mesmas motivações e acrescendo-se a sua facilidade de utilização e larga documentação disponível.
XML	O XML é um conjunto de regras semânticas normalizadas, que permitem uma linguagem de marcação de dados flexível e legível por máquinas e humanos.[29]	Utilizada para criar os dicionários de regras de tradução, pois permite fazê-lo de uma forma aberta e de fácil expansão.
Visual Paradigm for UML	Ferramenta de modelação UML, que inclui todos os modelos e diagramas definidos pelo UML [50].	Utilizada devido à sua simplicidade de geração dos vários diagramas UML utilizados (casos de uso, sequência e outros) no presente trabalho.

5.3 *MATLABER*

Como se viu no capítulo 4, a conversão dos códigos normalizados (em IL) é efetuada através de um sistema computacional desenvolvido no presente trabalho, o **Matlaber**. Na Figura 5.24, é possível ver o seu funcionamento, representada pelo seu ícone de aplicação.

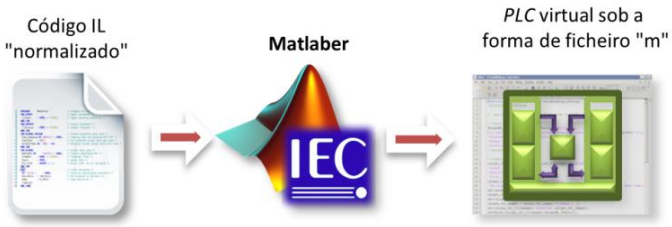

Figura 5.4 - Funcionamento do *Matlaber*

O **Matlaber** recebe, como dados de entrada, um ficheiro que contém o código do programa de CLP a converter. Este é processado e dá origem a um ficheiro ".m" para integração no *Matlab*. Pelo interface do **Matlaber** (Figura 5.5), pode observar-se como proceder a uma conversão. Escolhendo a origem dos ficheiros a converter, é apresentada uma lista de ficheiros passíveis de serem convertidos. O utilizador seleciona então qual (ou quais) dos ficheiros deseja converter. O destino dos ficheiros convertidos é gerado automaticamente a partir da origem, mas pode ser modificado pelo utilizador. Resta apenas acionar a conversão do(s) ficheiro(s) selecionado(s).

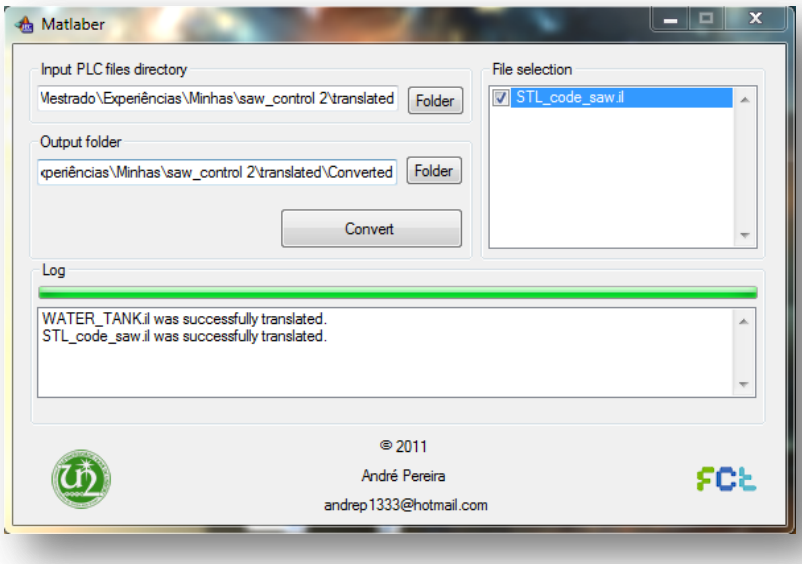

Figura 5.5 - Interface do *Matlaber*

5.3.1 ESTRUTURA DE DADOS DO *MATLABER*

Aquando do processo de conversão, a análise sintática resulta em código e informações para serem escritas para o ficheiro de saída. Internamente, os dados são escritos para uma estrutura de dados, como ilustrada na Figura 5.6. São estes os dados que são ultimamente convertidos para um ficheiro de texto, no último passo do processo. Não há nenhuma forma de retenção dos dados de conversão (base de dados ou armazenamento temporário em ficheiros), sendo os ficheiros convertidos por um processo direto e não acumulativo, pelo que se houver alguma falha no sistema computacional, basta reiniciar o processo.

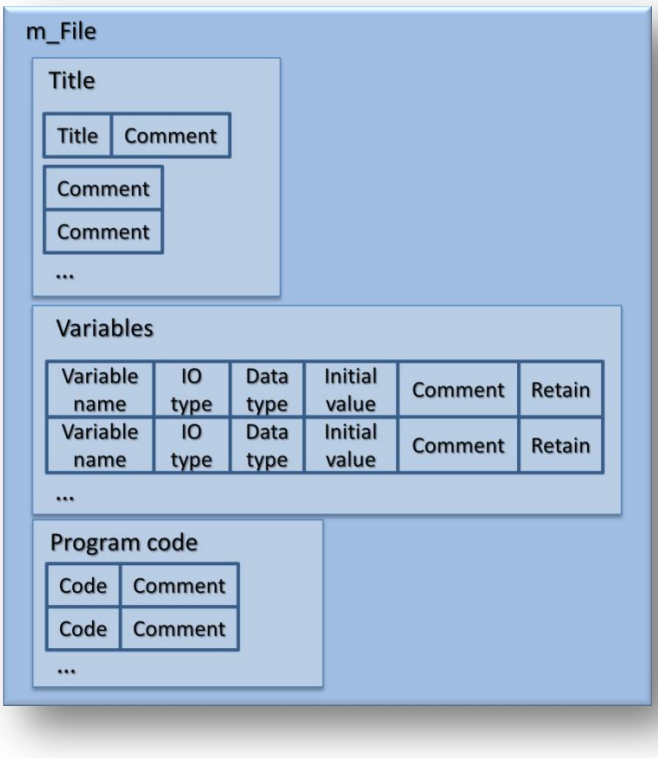

Figura 5.6 – Estrutura de dados de conversão para ficheiro ".m"

Como se pode observar, a estrutura de dados é constituída primariamente por 3 grupos: título (*Title*), variáveis (*Variables*) e código de programa (*Program Code*). O grupo *Title* possui uma subestrutura de dados com o título do programa e comentário associado, assim com uma listagem de comentários sem número definido. O grupo *Variables* possui uma listagem sem número definido de linhas de uma subestrutura de dados que contém: nome da variável, tipo de I/O, tipo de dados, valor inicial, comentário associado e indicador de propriedade de retenção. O

grupo *Program Code* possui uma listagem sem número definido de linhas de uma subestrutura de dados que contém código e comentário associado.

Estes campos irão sendo progressivamente preenchidos com o avançar da análise do processo de conversão.

5.3.2 FUNCIONAMENTO DO *MATLABER*

O processo de conversão do **Matlaber** recorre às várias classes indicadas no capítulo 4.3.1, cujo funcionamento e implementação são seguidamente explicados. Para fins de clareza, serão detalhados apenas os métodos mais importantes/relevantes.

5.3.2.1 CLASSE INTERFACE

Esta classe (Figura 5.7) tem associados 3 atributos.

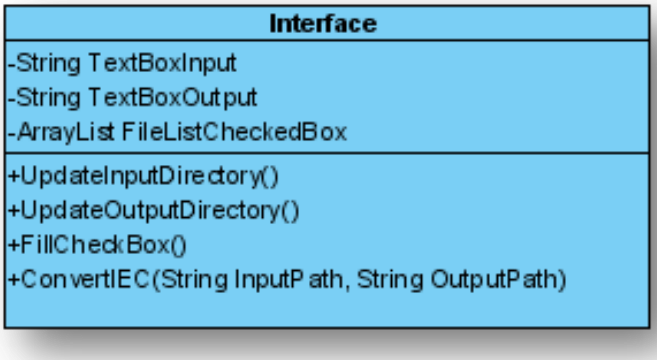

Figura 5.7 - Diagrama de classe do *Interface* do *Matlaber*

Estes atributos definem a chamada do processo de conversão e são introduzidos ou indicados pelo utilizador, nomeadamente:

- Origem dos ficheiros a converter (*TextBoxInput*).
- Destino dos ficheiros convertidos (*TexBoxOutput*).
- Listagem de ficheiros marcados para conversão (*FileListCheckedBox*).

Os métodos associados a esta classe são:

- *UpdateInputDirectory* – Método que atualiza internamente a diretoria de origem de ficheiros para conversão. É ativado sempre que há alguma alteração do atributo *TextBoxInput* pelo utilizador, recorrendo para isso à classe *Gestão de Dados*.
- *UpdateOutputDirectory* – Método que atualiza internamente a diretoria de destino de ficheiros convertidos. É ativado sempre que há alguma alteração do atributo *TexBoxOutput* pelo utilizador.
- *FillCheckBox* – Método que preenche a lista de ficheiros passíveis de serem convertidos, para seleção pelo utilizador, recorrendo para isso à classe *Gestão de Dados*.
- *ConvertIEC* – Método que chama a conversão por parte da classe *Conversor*.

5.3.2.2 CLASSE GESTÃO DE DADOS

Esta classe (Figura 5.8) tem associados 2 atributos.

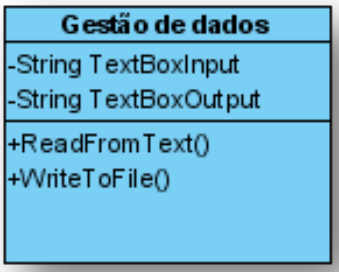

Figura 5.8 - Diagrama de classe da *Gestão de Dados* do *Matlaber*

Estes atributos são introduzidos ou indicados pelo utilizador através do interface, nomeadamente:

- Origem dos ficheiros a converter (*TextBoxInput*).
- Destino dos ficheiros convertidos (*TexBoxOutput*).

Estes 2 atributos definem os caminhos de chamada dos seus métodos. Os métodos associados a esta classe são:

- *ReadFromText* – Método que lê, para memória, o texto integral de um ficheiro a converter.

- *WriteToFile* – Método que escreve os ficheiros ".m" a partir da estrutura de dados e, de acordo com os requisitos indicados pelo conversor, gera também os ficheiros ".m" preconcebidos (descritos no capítulo 5.3.5).

5.3.2.3 CLASSE CONVERSOR

Esta classe (Figura 5.9) tem associados 2 atributos.

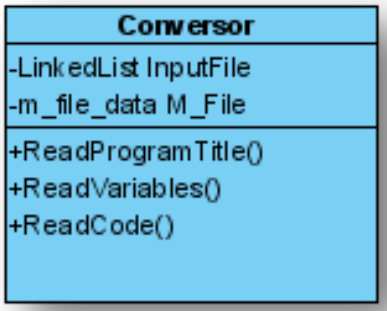

Figura 5.9 - Diagrama de classe do *Conversor* do *Matlaber*

São esses atributos:

- Ficheiro de entrada em memória (*InputFile*).
- Estrutura de dados em memória, onde é armazenada a conversão antes de ser escrito para ficheiro (*M_File*).

Os métodos associados a esta classe são:

- *ReadProgramTitle* – Método que processa a análise sintática do título do ficheiro a converter. O seu funcionamento está explicado mais detalhadamente no capítulo 5.3.3.1.
- *ReadVariable* – Método que processa a análise sintática das variáveis e sua declaração no ficheiro a converter. As variáveis suportadas pelo **Matlaber** podem ser consultadas no Anexo III. O funcionamento deste método está explicado mais detalhadamente no capítulo 5.3.3.2.
- *ReadCode* – Método que processa a análise sintática do código de programa do ficheiro a converter. O seu funcionamento está explicado mais detalhadamente no capítulo 5.3.3.3.

A sequência de conversão ativada pelo método *ConvertIEC*, da classe *Interface*, recorre aos 3 métodos associados à classe C*onversor*. O seu funcionamento está explicado em detalhe no capítulo 5.3.2.5.

5.3.2.4 FUNÇÕES/MÉTODOS DE SUPORTE

Os métodos detalhados na descrição de classes da secção prévia (5.3.2.3) representam apenas os mais importantes/relevantes, para fins de clareza. Várias funções/métodos de suporte tiveram que ser desenvolvidos para garantir o bom funcionamento do **Matlaber**, mantendo ao mesmo tempo uma implementação nivelada e hierárquica. Segue-se então uma descrição do papel e funcionamento das funções/métodos de suporte.

- *RemoveComment* – Separa o código útil do texto de comentário de uma dada linha de código, procurando pelo marcadores de comentário da linguagem IL: "(*Texto de Comentário*)".
- *RemoveIlegalChars* – Carateres como tabulações e quebras de linha são irrelevantes no contexto de análise sintática, sendo esta função utilizada para a substituição de carateres disruptivos do processo de análise sintática, por um carácter simples de um espaço " ".
- *RemoveSpaces* – Variáveis em *Matlab* têm que ter nomes que obedecem a certas regras, entre as quais que uma variável terá que ser uma sequência contínua de carateres, sem espaços. Para tal fim, esta função elimina os espaços de um dado nome de variável.
- *IsFileEmpty* – Verifica se o ficheiro indicado possui algum texto ou se se encontra vazio. Utilizado para negar o processo de conversão, caso o ficheiro não contenha dados.
- *GetFileFromAssembly* – São utilizados alguns elementos no sistema computacional, como imagens do interface e códigos preconcebidos (ver capítulo 5.3.5) que estão embutidos sob a forma de *assemblies*[9], dados embutidos no binário da aplicação.
- *ValidateAssemblies* – As *assemblies* são suscetíveis de corrupção, o que poderá originar um resultado de conversão incoerente com o previsto. Para evitar tais situações, todas as *assemblies* são verificadas através de um processo de *hashing*[10] e comparação. Isto garante, probabilisticamente, a integridade de todas as *assemblies*. Esta verificação ocorre cada vez que o sistema computacional é iniciado.

[9] Dados encapsulados na aplicação.

[10] Processo de geração de uma sequência de valores binários que representam não deterministicamente uma sequência binária original de tamanho indefinido.

- *SortVariables* – Ordena as variáveis da estrutura de dados pelo seu nome, necessário por motivos de coerência de dados, aquando da declaração das variáveis em *Matlab*.
- *SeparateInstFromArg* – Função de suporte da análise sintática de código, separa uma *string* pelo primeiro carácter de espaço (" "). Este funcionamento é indicado para a separação de instrução e argumento da linguagem IL.

5.3.2.5 SEQUÊNCIA DE CONVERSÃO

O diagrama de sequência da Figura 5.10 ilustra o funcionamento interno do **Matlaber** mais detalhadamente, aquando de uma conversão de 2 ficheiros sequencialmente. O conversor suporta (n) ficheiros para conversão sequencial, mas para ilustrar o seu funcionamento de uma forma clara, o diagrama de sequência da Figura 5.10 apenas ilustra 2.

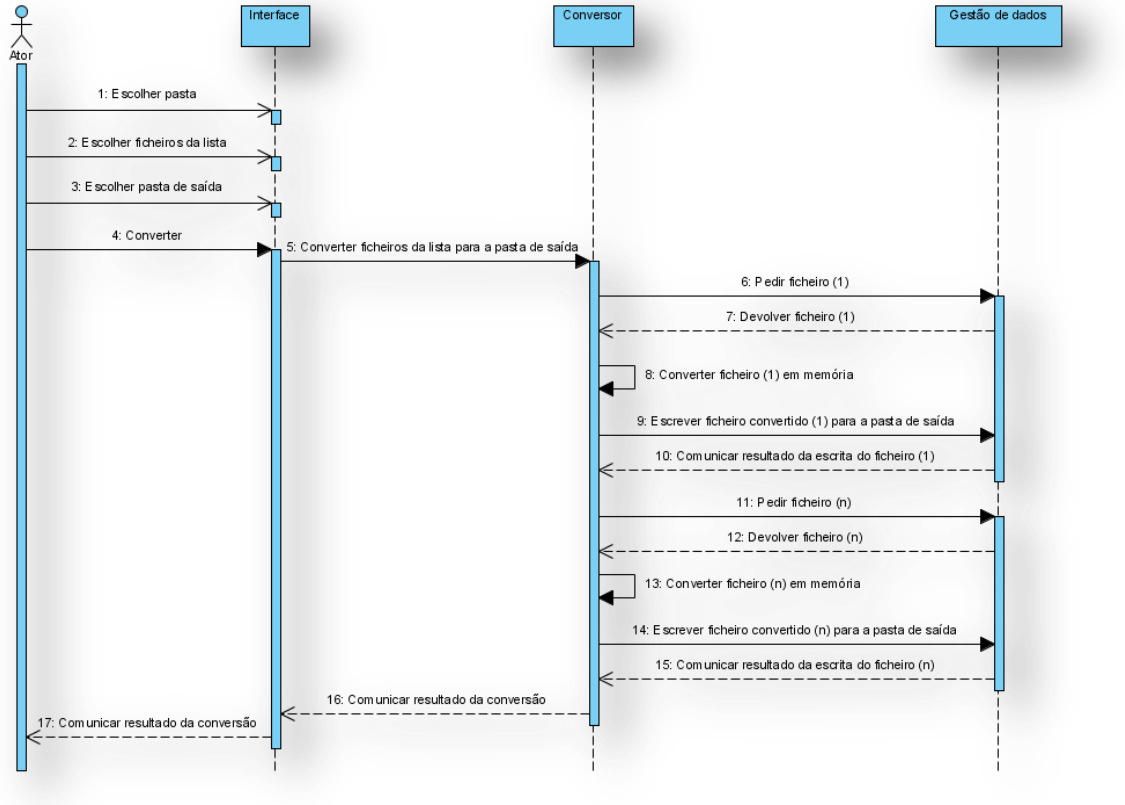

Figura 5.10 - Diagrama de sequência de uma conversão de 2 ficheiros no *Matlaber*

Segue-se uma descrição dos vários passos desta conversão de 2 ficheiros:

1. Através da *Interface*, escolhendo a origem dos ficheiros a converter, é apresentada uma lista de ficheiros passíveis de serem traduzidos.

2. O utilizador seleciona então qual (ou quais) dos ficheiros deseja converter.
3. O destino dos ficheiros convertidos é gerado automaticamente a partir da origem, mas pode ser modificado pelo utilizador.
4. O utilizador aciona a conversão dos ficheiros selecionados.
5. Acionar a conversão dos ficheiros selecionados no *Conversor*.
6. O conversor pede o 1º ficheiro indicado na lista à *Gestão de Dados*.
7. A *Gestão de Dados* devolve ao *Conversor* o ficheiro pretendido sob a forma de uma listagem de texto em memória.
8. O processo de conversão é ativado. Este está detalhadamente explicado no próximo capítulo (5.3.3).
9. O ficheiro convertido é escrito a partir da estrutura de dados em memória gerada pelo processo de conversão. Este processo está detalhadamente explicado no capítulo 5.3.7.
10. Resposta de sucesso da escrita do ficheiro.

Os passos 11 a 15 são análogos aos passos 6 a 10, mas para o 2º ficheiro indicado na lista.

16. Findas ambas as conversões e caso não hajam erros, o *Conversor* indica à *Interface* o sucesso do processo.
17. A Interface informa o utilizador, através de uma caixa de mensagem, do sucesso das conversões.

De notar qualquer erro numa conversão interromperá imediatamente (apenas) o processo do ficheiro a ser convertido, mostrando uma mensagem de erro adequada, com o sistema computacional a informar o utilizador dos resultados específicos de cada ficheiro, no final.

5.3.3 PROCESSO DE CONVERSÃO

O processo de conversão é a fase crucial do trabalho desenvolvido. É executado em 3 fases distintas (Figura 5.11) através de 3 métodos diferentes, como indicados na descrição da classe *Conversor* (capítulo 5.3.2.3).

Os 3 métodos são chamados ordenadamente e aplicam uma análise sintática que escreve o seu resultado para a estrutura de dados. São eles:

- *ReadProgram Title* – Ler Título.
- *ReadVariables* – Ler Variáveis.
- *ReadCode* – Ler Código.

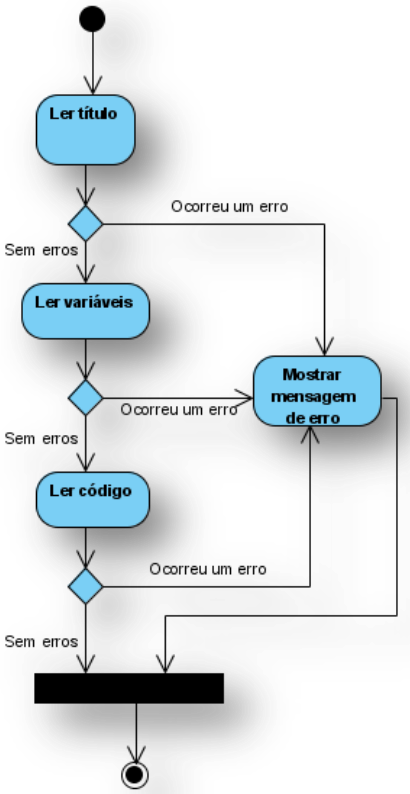

Figura 5.11 – Diagrama de atividades com as 3 fases do processo de conversão do *Matlaber*

Todo o processo de análise e conversão segue o funcionamento típico de um *parser* – uma análise sintática, linha a linha, que procura palavras e/ou carateres chave para interpretação. Este processo é diferente para cada fase, pelo que é explicado em detalhe para cada uma nos capítulos seguintes.

5.3.3.1 READPROGRAMTITLE - *LER TÍTULO*

O IEC 61131-3 tem a linha de título de programa ou POU bem definida (ver exemplo da Figura 5.12), que será sempre a primeira linha do código.

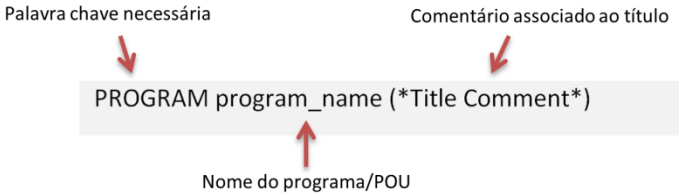

Figura 5.12 - Formato do título em IL

O conversor retira daqui o título do POU e escreve-o para a estrutura de dados, que será o nome final da função sob a forma de ficheiro ".m". Este formato terá de ser estritamente seguido, caso contrário o **Matlaber** indicará um erro de sintaxe na linha do título. Uma listagem completa do código de implementação deste método pode ser consultada no Anexo IV.

5.3.3.2 READVARIABLES – LER VARIÁVEIS.

A declaração de variáveis é bem definida pela norma, como no exemplo da Figura 5.13, onde estão indicadas algumas palavras-chave ou indicativos de propriedades das variáveis. Estes campos são:

- Tipo de I/O – Indicado pela linha de início de declaração de variáveis.
- Comentário associado à variável – Cada linha de código poderá ter sempre um texto de comentário associado, não sendo a declaração de variáveis uma exceção.
- Nome da variável – Pode ser qualquer conjunto de carateres sem espaços.
- Campo opcional de retenção – Indica se os valores em memória devem ser mantidos mesmo quando o CLP é desligado.
- Acesso a endereço físico – Permite o acesso direto a endereços fixos, isto é, valores de I/O.

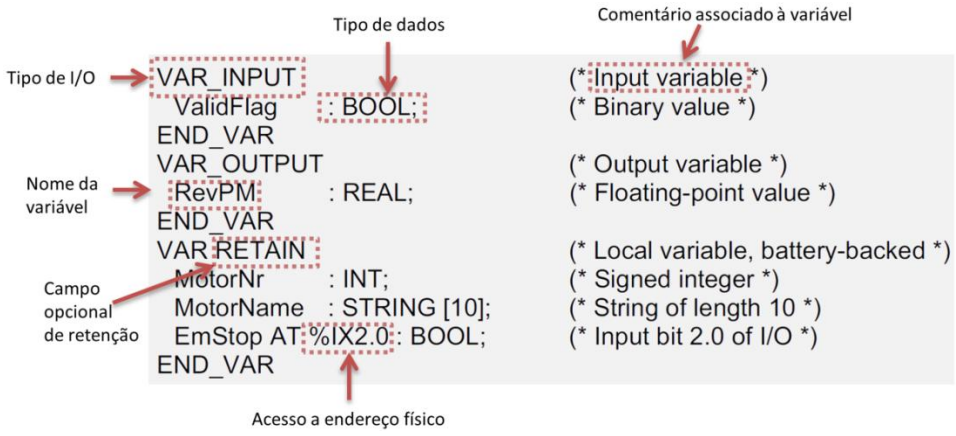

Figura 5.13 - Exemplo de declaração de vários tipos de variáveis [13]

O **Matlaber** lê os vários campos aplicáveis a cada variável, como o tipo de I/O, tipo de dados, valor inicial e parâmetro de retenção e escreve-os para o modelo de dados. Se não for indicado valor inicial, o **Matlaber** atribui automaticamente um, de acordo com o tipo de dados. Os tipos de variáveis suportados pelo trabalho desenvolvido estão detalhados no Anexo III. Apesar da

sua variedade, quando convertidas para *Matlab*, resumem-se a 3 tipos: booleanas, reais[11] e *string*[12]. Uma listagem completa do código de implementação deste método pode ser consultada no Anexo V.

Para o funcionamento correto do código *Matlab*, as variáveis internas do CLP têm que manter-se entre chamadas do programa. De lembrar que o programa de controlo estará implementado sob a forma de uma função *Matlab*, chamada repetidamente, não tendo nenhuma memória interna associada que se retenha entre chamadas. Portanto, os valores de memória do CLP têm que ser declarados como variáveis globais do *Matlab*, acessíveis externamente e que se retêm entre chamadas do programa de controlo. Também é necessário inicializar essas variáveis com os valores indicados pelo código IL original.

A solução, para este problema, foi a criação de uma função *Matlab* paralela, dedicada à declaração global das variáveis e sua inicialização correspondente. Esta função é criada sob a forma de um ficheiro ".m" de inicialização, um *start file*. O seu nome é sempre criado automaticamente a partir do nome do programa de controlo. Tomemos como exemplo, um programa de controlo chamado "boolean_control.m", terá um ficheiro de inicialização "boolean_control_startfile.m". Este ficheiro de inicialização garante que as variáveis existem antes de serem utilizadas pelo programa de controlo, assim como garante a sua retenção entre ciclos de chamada do CLP, caso contrário este não funcionaria. Um exemplo de um ficheiro de inicialização pode ser consultado na Figura 5.14. Os nomes das variáveis também sofrem uma transformação no processo de conversão, mais notavelmente os endereços de I/O diretos, que se transformam para a sua versão em *Matlab* como, por exemplo, "% I 0.0" para "I0_0". Uma transformação semelhante é efetuada em todas as variáveis cujos nomes não obedeçam a regras de nomenclatura do *Matlab*.

Na Figura 5.14 é de notar a declaração das variáveis como globais. Todas as variáveis, cujo valor seja necessário manter entre cada ciclo de execução do CLP, serão assim declaradas, sendo elas:

- Variáveis globais (*global*).
- Variáveis de saída (*output*).
- Variáveis com o parâmetro de retenção ativo (*retention*).

[11] Valores numéricos racionais.

[12] Sequência de carateres.

Não só serão declaradas no *start file* mas também no ficheiro ".m" principal, o que garante o funcionamento pretendido, de acordo com o funcionamento interno do *Matlab*.

```
%-This file declares all the variables used and sets their initial value---
%------Run this before running the main m file----------------------------
%-------------------------------------------------------------------------

%------------------------variables' initialization------------------------
global DG_0 DG_1 DG_2 DG_3
DG_0=0;
DG_1=0;
DG_2=0;
DG_3=0;
%-------------------------------------------------------------------------
```

Figura 5.14 - Exemplo de um ficheiro de inicialização *start file*

5.3.3.3 READCODE - LER CÓDIGO

Linha a linha, o **Matlaber** vai processando cada instrução do código IL, onde a ideia principal é a de uma conversão para equação. O resultado será progressivamente escrito para a estrutura de dados. Uma listagem completa do código de implementação deste método pode ser consultada no Anexo VI.

Dado o funcionamento por acumulador do IL (como referido no capítulo 5.1.4), decidiu-se adotar uma escrita equacional para conversão. O código original que inclua o recurso ao acumulador é convertido numa equação matemática simples. A Figura 5.15 ilustra um exemplo simples de conversão para escrita equacional.

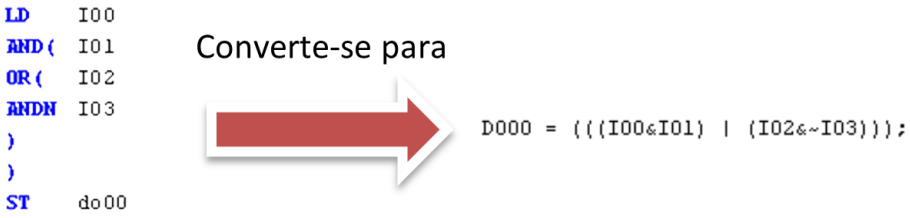

Figura 5.15 - Exemplo de uma conversão equacional

De notar que 7 linhas de código se converteram em apenas uma, bastante legível por humanos. Ao contrário do funcionamento do IL, o resultado numérico da operação de cada linha de código não é escrito para o acumulador, mas antes as operações matemáticas associadas, criando uma espécie de **acumulador equacional** (este acumulador equacional será referido

várias vezes aquando da explicação mais pormenorizada do processamento de cada instrução). Quando surgir uma instrução, que faça recurso ao acumulador numérico, este será escrito de uma só vez para as variáveis de saída sob a forma da equação matemática correspondente ao código original. Esta escrita é muito mais acessível e legível para humanos, permitindo uma análise rápida do funcionamento subjacente, contribuindo para uma celeridade de análise e testes simulados, que são o objetivo final do trabalho apresentado.

Para o processamento das variadas instruções (ou "operadores") definidas pelo IEC 61131-3, é importante referir que o IL é uma linguagem de baixo nível, possuindo um alcance de instruções bem definido. Para o presente trabalho, foi desenvolvido suporte para as instruções do IL indicadas na Tabela 5.2.

Tabela 5.2 – Instruções IL suportadas pelo trabalho desenvolvido

Instrução (operador)	Modificadores de instrução	Descrição
LD	N	Definir acumulador
ST	N	Guardar o acumulador
S		Definir como verdadeiro (booleano)
R		Definir como falso (booleano)
AND	N, (	Conjunção binária
OR	N, (	Disjunção binária
XOR	N, (	Disjunção binária exclusiva
ADD	(	Soma aritmética
SUB	(	Subtração aritmética
MUL	(	Multiplicação aritmética
DIV	(	Divisão aritmética
GT	(	Comparação "maior"
GE	(	Comparação "maior ou igual"
EQ	(	Comparação "igual"
NE	(	Comparação "diferente"
LE	(	Comparação "menor ou igual"
LT	(	Comparação "menor"
JMP	C, N	Saltar para
CAL	C, N	Chamar sub-rotina
RET	C, N	Voltar da sub-rotina
)		Devolver valor do acumulador

Os modificadores da instrução acrescentam variações possíveis, como por exemplo *LDN*, que é um *LD* negado[13]. Às instruções apresentadas na Tabela 5.2, acrescenta-se ainda a declaração de *labels*[14], que não é uma instrução em si, mas é utilizada agregando dois pontos (":") ao nome da *label* como, por exemplo, "LABEL ABC:".

Segue-se uma descrição do propósito e método de conversão de cada instrução.

- Instruções de carregamento e armazenamento:
    - LD (*load*) – carregar valor da variável do argumento para o acumulador. O valor é carregado para o acumulador equacional.
    - ST (*store*) – armazenar valor do acumulador para a variável do argumento. O valor é armazenado para o acumulador equacional.
- Instruções de elementos biestáveis:
    - S (*set*) – Definir a variável do argumento (booleano) como verdadeira. O valor é definido como verdadeiro.
    - R (*reset*) – Definir variável do argumento (booleano) como falsa. O valor é definido como falso.
    - ST (*store*) – armazenar valor do acumulador para a variável do argumento. O valor é armazenado para o acumulador equacional.
- Instruções de lógica booleana (para cada uma destas instruções, é adicionada a operação correspondente em *Matlab* ao acumulador equacional):
    - AND (*and*) – Conjunção lógica da variável do argumento com o acumulador.
    - OR (*or*) – Disjunção lógica da variável do argumento com o acumulador.
    - XOR (*exclusive or*) – Disjunção lógica exclusiva da variável do argumento com o acumulador.
- Instruções de operações aritméticas (para cada uma destas instruções, é adicionada a operação correspondente em *Matlab* ao acumulador equacional):
    - ADD (add) – Adicionar valor da variável do argumento ao acumulador.
    - SUB (*subtract*) - Subtrair valor da variável do argumento ao acumulador.
    - MUL (*multiply*) - Multiplicar valor da variável do argumento com o acumulador.
    - DIV (*divide*) - Dividir o acumulador pelo valor da variável do argumento.

[13] Valor lógico invertido.

[14] As *labels*, neste contexto, referem-se a marcadores de posição no código definidos pelo utilizador.

- Instruções de comparação (para cada uma destas instruções, é adicionada a operação correspondente em *Matlab* ao acumulador equacional):
    - GT (*greater than*) – Maior que o acumulador.
    - GE (*greater or equal*) – Maior ou igual ao acumulador.
    - EQ (*equal*) – Igual ao acumulador.
    - NE (*not equal*) – Diferente do acumulador.
    - LE (*less or equal*) – Menor ou igual que o acumulador.
    - LT (*less than*) – Menor que o acumulador.
- Instruções de fluxo de programa (para cada uma destas instruções, ver abaixo uma explicação mais detalhada):
    - JMP (*jump*) – Salto direto ou condicional para labels ou linhas específicas.
    - *Labels* – Marcadores de código para saltos de fluxo.
    - CAL (*call*) – Chamada de outros POU.
    - RET (*return*) – Voltar ao POU de onde foi chamado o último CAL.

5.3.4 LÓGICA DE FLUXO DE PROGRAMA

A lógica de fluxo de programa dos CLP é baseada no método antiquado (entretanto substituído em linguagens de programação modernas [51]) de saltos diretos de código (*goto*[15]) para linhas ou *labels*, com as instruções *jump*.

Dada a já referida natureza simbólica da tradução, saltos para linhas diretas ou mesmo saltos para *labels* entre manipulação de valores não podem ser diretamente suportados. Não há correspondência de linhas/código e, mesmo que fosse criada uma tabela de correspondência, um código CLP de muitas linhas pode traduzir-se para apenas uma (como no exemplo da Figura 5.16), portanto deixa de fazer sentido o uso deste método.

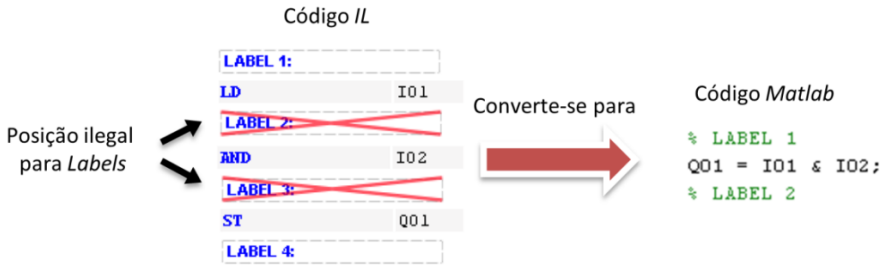

Figura 5.16 - Exemplo de uso de *labels, com* posições "legais" e código convertido

[15] O *goto* é uma instrução de salto unidirecional para uma dada linha de código.

Esta limitação é uma desvantagem reconhecida que advém do método de conversão simbólica adotado neste trabalho. No entanto, é sempre possível contornar esta limitação, reescrevendo o código a ser traduzido com a inclusão de *labels* em locais "legais" (Figura 5.16) e executando os saltos para essas *labels*, em vez de para linhas.

O *Matlab* não suporta nenhuma funcionalidade de saltos diretos pelo que é necessário providenciar uma funcionalidade semelhante para garantir o máximo de fidelidade do código convertido. Para replicar o funcionamento do *goto*, foi utilizada uma biblioteca pública e aberta que emula essa função – "*MATLAB Goto Statement*" por Husam Aldahiyat [52], cujo uso para esta dissertação foi explicitamente autorizado pelo autor. Esta funcionalidade é necessária para executar um fluxo de programa fiel ao original e é emulada sob a forma de um ficheiro ".m", sendo chamada tal como uma função de *Matlab*. Um exemplo da sua utilização (retirado da documentação da biblioteca "*MATLAB Goto Statement*") pode ser consultado na Figura 5.17.

```
1   % this is a while loop using goto()
2
3   a = 5;
4
5   % LABEL START
6   a = a - 1;
7   disp(a)
8   if a > 0
9       goto('START');
10      return
11  end
```

Figura 5.17 - Exemplo de código com recurso à função *GOTO*

Este exemplo consiste num contador regressivo de 5 até 0. A chamada do *goto* está condicionada ao valor da variável "a" e não será executada quando "a" for maior que zero. A declaração da *label* "START" é de acordo com o estipulado na biblioteca, sob a forma de um comentário de código *Matlab*. A instrução *return* faz parte do uso da biblioteca "*MATLAB Goto Statement*", sendo sempre necessária a sua inclusão após uma chamada à função *goto*.

Como qualquer função de *Matlab* não nativa, esta tem que estar na mesma diretoria de trabalho que a função que a chama. Como tal e para garantir a funcionalidade, um ficheiro "goto.m" é gerado para a mesma diretoria de saída, cada vez que o ficheiro ".m" do CLP é gerado.

O suporte para as instruções do tipo "*CAL*" e "*RET*" é oferecido através de uma reescrita da linha, tirando os casos especiais (chamadas de blocos funcionais). Um exemplo desta conversão pode ser consultado na Figura 5.18.

A instrução "*CAL*" é convertida numa chamada de *Matlab* com os parâmetros de chamada preservados. Inclui ainda a chamada dos POU base definidos pela norma. "*RET*" é chamado na sua função equivalente de *Matlab* (*return*), mas não antes do vetor de dados de saída da função atual ser atualizado com os valores correspondentes.

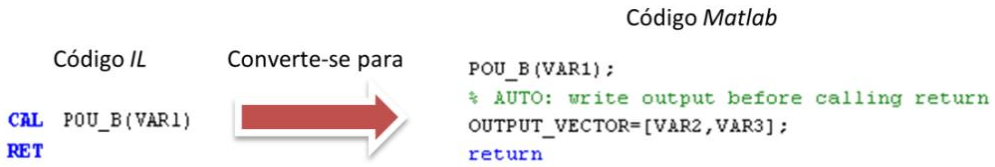

Figura 5.18 - Exemplo de conversão de "*calls*" e "*returns*"

Este funcionamento é coerente com o formato do ficheiro ".m", como definido no capítulo 5.1.3.

5.3.5 CHAMADA DE BLOCOS FUNCIONAIS INCLUÍDOS NA NORMA

A norma IEC 61131-3 inclui vários blocos funcionais, sob a forma de POU, que deverão estar disponíveis em todos os equipamentos disponíveis. Estes incluem *Flip-Flop*s, contadores e temporizadores. Para uma lista completa, consultar o Anexo II. Para a sua implementação se manter fidedigna e coerente com a norma IEC 61131-3, a abordagem tomada para estes blocos funcionais foi a de a criação de ficheiros ".m" que possam ser chamados tal como o são em código IL. Sendo funções *Matlab*, não possuirão retenção de memória, pelo que os dados de estado serão armazenados na própria variável declarada do bloco funcional, em campos extra. Assim, poderá haver várias iterações que utilizem o mesmo bloco funcional.

Para este trabalho, foram desenvolvidos os blocos funcionais ***TON*** (*TimeOnTimer*), ***CTU*** (*CounterUp*) e ***CTD*** (*CounterDown*), que sobejam os requisitos para a execução dos programas de controlo de validação a que o trabalho se propõe. As suas implementações podem ser consultadas nos anexos VII, VIII e IX, respetivamente.

5.3.6 RETENÇÃO DE COMENTÁRIOS

Dado que o objetivo principal do presente trabalho é possibilitar testes simulados para estudo, fez-se questão de manter intactos quaisquer comentários que estivessem no código de programa original. Isto ajudará a localizar marcadores ou códigos problemáticos do código original,

identificando-os no código convertido e ajudando nos processos de estudo, análise e *debug*. Um exemplo da retenção de comentários numa conversão pode ser consultado na Figura 5.19.

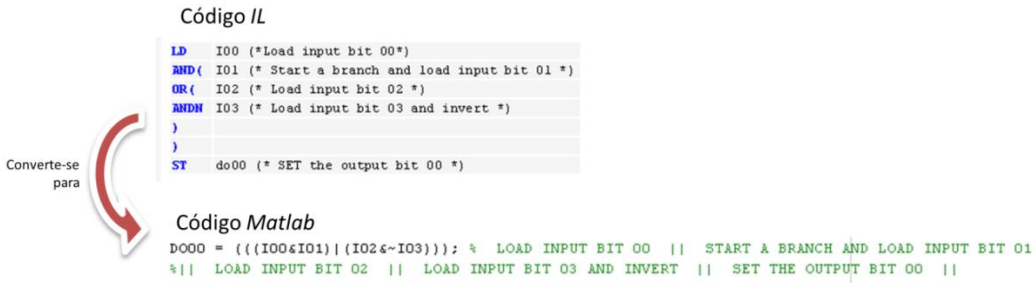

Figura 5.19 - Exemplo de conversão com retenção de comentários

Os comentários de cada linha do código original são copiados para o modelo de dados e apagados antes da análise sintática de código processar a linha.

5.3.7 ESCRITA PARA FICHEIRO ".M" DE SAÍDA - *WRITETOFILE*

A fase de escrita para ficheiro utiliza as informações na estrutura de dados para gerar o ficheiro de saída, no formato predefinido pelo modelo de função *Matlab*, como ilustrado na Figura 5.20.

$$function[output] = \text{Nome\_do\_POU}(clock\_in, di_1, \ldots, di_n, \ldots, ai_1, \ldots, ai_m)$$
$$\ldots$$
$$\begin{pmatrix} \text{Programa de controlo PLC} \\ \text{em linguagem Matlab/Simulink} \end{pmatrix}$$
$$\ldots$$
$$output = [do_1, \ldots, do_p, \ldots, ao_1, \ldots, ao_q]$$

Figura 5.20 - Formato geral do ficheiro ".m"

O número de entradas/saídas define os argumentos de entrada/saída da função. As entradas di_1 a di_n representam as "n" entradas digitais e ai_1 a ai_m representam as "m" entradas analógicas. As saídas do_1 a do_p representam as "p" saídas digitais e ao_1 a ao_q representam as "q" saídas analógicas. É importante referir que o tamanho do bloco de multiplexação de entrada (**b**) é igual a "n+m+1" e o do bloco de desmultiplexação de saída (**c**) é igual "p+q". O *clock_in* é sempre uma variável de entrada, representa o valor da unidade temporal atual. É um valor utilizado internamente pela função e pelas chamadas recursivas a outros POU, não sendo influenciado pelo código de programa.

Para além do ficheiro principal, que representa o programa de controlo principal do CLP, são ainda gerados outros ficheiros ".m" de suporte, tais como:

- *Start file* – Utilizado para inicializar as variáveis necessárias no ambiente *Matlab*/*Simulink*, como indicado no capítulo 5.3.3.2.
- *Goto file* – Utilizado para replicar o fluxo original do programa de controlo, como indicado no capítulo 5.3.4**Erro! A origem da referência não foi encontrada.**.
- *POU files* – como indicado no capítulo 5.3.5. Só são gerados os POU que sejam utilizados no programa de controlo.

Tanto os POU *files* como o *Goto file* encontram-se integralmente embutidos no binário do sistema computacional, sob a forma de *assemblies*. São portanto copiados para os ficheiros gerados quando relevante, não sendo necessário por isso a sua disponibilidade através de outros meios. Como já referido no capítulo 5.3.2.4, a sua integridade é verificada cada vez que o sistema computacional é iniciado.

5.3.8 EXEMPLO ILUSTRATIVO DE CONVERSÃO

Para se poder visualizar melhor o resultado prático de uma conversão, apresenta-se um código de programa de controlo CLP em IL (Figura 5.21), que se pretende converter, segundo o método proposto, para fins ilustrativos.

Este é um programa abstrato sem significado real, como demonstração das capacidades de conversão. Controla uma lâmpada que é ativada de acordo com operações de lógica de um conjunto de entradas. Como qualquer programa em IL, encontra-se escrito em 3 partes diferentes, seguidamente descritas:

- A linha 1 contém a declaração do título do programa/POU. O nome do programa/POU é indispensável pois é o nome pelo qual outros POU o podem chamar.
- As linhas 3 a 12 são de declaração de variáveis. Esta declaração é independente da linguagem de programação do IEC 61131-3 escolhida.
- As linhas 14 a 20 são o troço funcional do programa, onde as saídas de controlo são geradas. É aqui que reside o maior interesse do trabalho desenvolvido, que tem como requisito manter o funcionamento original do programa de controlo de CLP.

```
1   PROGRAM Boolean_Test (*Title Comment*)
2
3   VAR_INPUT
4   % I 0.0 :BOOL:=false;
5   % I 0.1:BOOL;
6   % I 0.2:BOOL;
8   % I 0.3:BOOL;
9   END_VAR
10  VAR_OUTPUT
11  Lightbulb_control :bool; (*Lightbulb!*)
12  END_VAR
13
14  LD I00 (* Load input bit 00 *)
15  AND( I01 (* Start a branch and load input bit 01 *)
16  OR( I02 (* Load input bit 02 *)
17  ANDN I03 (* Load input bit 03 and invert *)
18  )
19  )
20  ST Lightbulb_control
21
22  END_PROGRAM
```

Figura 5.21 - Código exemplo a converter

Quando convertido com recurso ao **Matlaber**, obtém-se o código da Figura 5.22 e Figura 5.23. Podem-se observar vários elementos, desta conversão, como detalhados no capítulo corrente, nomeadamente:

- A formação do ficheiro ".m" de acordo com o modelo indicado no capítulo 5.1.3, incluindo entradas e saídas.
- Inclusão automática do valor de entrada de relógio, o "`CLOCK_IN`".
- A declaração da variável de saída ("`LIGHTBULB_CONTROL`"), como variável global do *Matlab*, como indicado no capítulo 5.3.3.2.
- A conversão simbólica, que resume todo o programa de controlo a uma linha de equação matemática booleana.
- Retenção de todos os comentários do código original, como referido no capítulo 5.3.6.

```
function
[OUTPUT_VECTOR]=BOOLEAN_TEST_SIMPLE(CLOCK_IN,I0_0,I0_1,I0_2,I0_3)
%TITLE COMMENT
%Converted by IEC-Matlab converter by Andre' Pereira
%08-07-2011 01:46:45 Hora padrÃ£o de GMT

global LIGHTBULB_CONTROL
%--------------------Variables' comments-----------------------
%output LIGHTBULB_CONTROL's comment: LIGHTBULB!
%--------------------------------------------------------------
LIGHTBULB_CONTROL = (((I00&I01)|(I02&~I03))); %  LOAD INPUT BIT 00
%||  START A BRANCH AND LOAD INPUT BIT 01  ||  LOAD INPUT BIT 02
%||  LOAD INPUT BIT 03 AND INVERT  ||

%Output generation
OUTPUT_VECTOR=[LIGHTBULB_CONTROL];
```

Figura 5.22 - Função *Matlab* resultante da conversão exemplo

Para além das observações anteriores relativamente ao código principal, também foi gerado um *start file* (Figura 5.23), como indicado no capítulo 5.3.3.2, onde se pode observar:

- A declaração da variável de saída ("`LIGHTBULB_CONTROL`"), como variável global do *Matlab*, como indicado no capítulo 5.3.3.2.
- A inicialização a verdadeiro da variável de saída "`LIGHTBULB_CONTROL`".

```
%This file declares all the variables used and sets their
%initial value
%------Run this before running the main m file-------------
%----------------variables' initialization-----------------
global LIGHTBULB_CONTROL
LIGHTBULB_CONTROL=1;
```

Figura 5.23 - Função de inicialização da conversão exemplo - *start file*

5.4 *UNIFIL*

Como se viu no capítulo 4, a "normalização" dos códigos proprietários é efetuada com recurso a um ficheiro de dicionário correspondente à linguagem CLP proprietária. Na Figura 5.24, é possível ver este funcionamento através da aplicação desenvolvida, representada pelo seu ícone de programa.

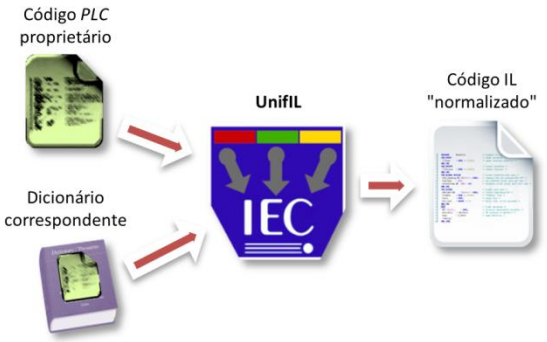

Figura 5.24 - Funcionamento do *UnifIL*

O **UnifIL** recebe, como dados de entrada, um ficheiro que contém código de CLP proprietário a traduzir e um ficheiro de dicionário de regras de tradução correspondente a essa linguagem proprietária. Este é traduzido, dando origem a um ficheiro com código IL normalizado de acordo com o IEC 61131-3 com o mesmo funcionamento. Pelo interface do **UnifIL** (Figura 5.25), pode-se observar como proceder a uma conversão.

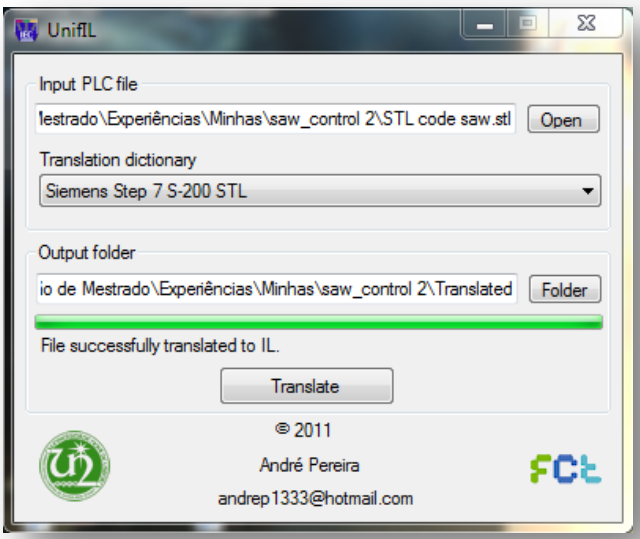

Figura 5.25 - Interface no *UnifIL*

O utilizador começa por indicar o ficheiro a traduzir, sendo o destino dos ficheiros traduzidos gerado automaticamente a partir da origem, mas pode ser modificado pelo utilizador. É necessário ainda escolher qual o dicionário de regras de tradução a utilizar, dos disponíveis na lista de dicionários. Resta apenas acionar a tradução do ficheiro selecionado.

5.4.1 ESTRUTURA DE DADOS DO *UNIFIL*

Aquando da tradução, a análise sintática resulta em código e informações para serem escritas para o ficheiro traduzido de saída. Internamente, os dados são escritos para uma estrutura de dados, descrita na Figura 5.26. Estes dados é que são ultimamente convertidos para texto simples, no último passo do processo, onde são escritos para o ficheiro de saída. Não há nenhuma forma de retenção dos dados de tradução (base de dados ou armazenamento temporário em ficheiros), sendo os ficheiros convertidos por um processo direto e não acumulativo, pelo que se houver alguma falha no sistema computacional, basta reiniciar o processo.

Como se pode observar na Figura 5.26, a estrutura de dados é constituída primariamente por 3 grupos, título (*Title*), variáveis (*Variables*) e código de programa (*Program Code*).

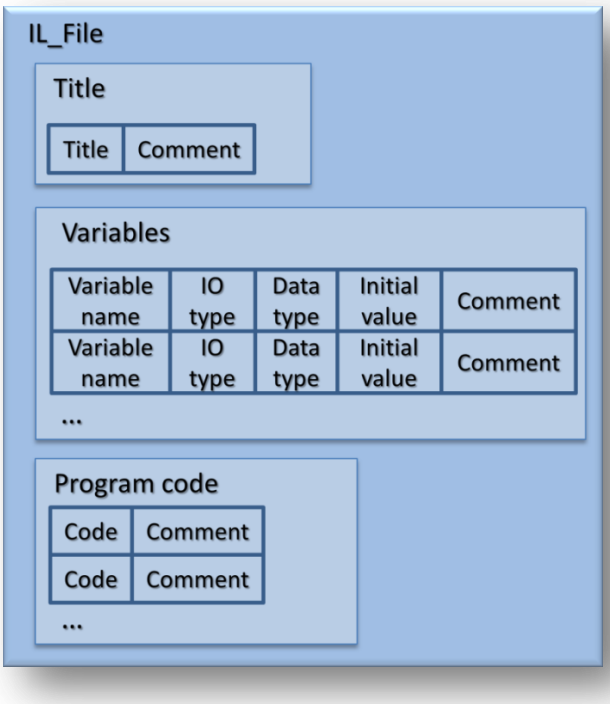

Figura 5.26 - Estrutura de dados interna de um ficheiro traduzido

O grupo *Title* possui uma subestrutura de dados com o título do programa de CLP e o comentário associado, caso exista. O grupo *Variables* possui uma listagem de número indefinido de linhas de uma subestrutura de dados que contém: o nome da variável, o tipo de I/O, tipo de dados, valor inicial e comentário de declaração. O grupo *Program Code* possui uma listagem de número indefinido de linhas de uma substrutura de dados que contém código e comentário associado.

Todos estes campos irão sendo progressivamente preenchidos com o avançar do processo de tradução. Para além da estrutura de dados, o **UnifIL** faz também recurso a ficheiros de dicionários de regras de tradução.

5.4.2 DICIONÁRIOS DE REGRAS DE TRADUÇÃO

O tradutor segue regras definidas externamente por um dicionário de regras de tradução, contendo todos os elementos necessários à tradução do código proprietário. O conteúdo do dicionário indica ao tradutor que ação tomar quando dadas condições são encontradas, sendo os elementos de regras precisamente as instruções de ação. No âmbito deste trabalho, foram desenvolvidas funcionalidades suficientes para a tradução de vários programas de CLP (ver capítulo 6) que são descritas abaixo:

- Regras gerais – suporta adição de texto predefinido ao início do código de saída, sob a forma de código e/ou comentário.
- Tipos de variáveis – permitem reconhecer qualquer tipo de variável, desde que a correspondente em IL seja uma de: *bool*, *integer* ou *string*. O seu tipo de I/O pode ser de entrada, saída ou variável interna (global). Tem capacidade de reconhecer informações como o tipo de I/O através do formato de nomes variáveis, assinalando as partes únicas do nome com asterisco (*). Como exemplo do dicionário STL**, I*.***, corresponde a qualquer variável de entrada cujo nome pode ser **I0.0**, **I1.0**, **I1.1**, etc…
- Marcadores de comentários – suportam qualquer conjunto de carateres como marcador de início ou fim de comentários, para além do fim da linha de código como o fim dos comentários.
- Instruções – indicam todas as instruções de CLP reconhecidas, assim como o seu código de saída correspondente e regras específicas a aplicar. Foram implementadas todas as instruções de lógica booleana, a maior parte das de andamento de programa e algumas de temporizadores e acumuladores.

O formato dos dicionários, em XML, pode ser descrito em pormenor através do DER (Diagrama de Entidades e Relações) do seu *schema* (Figura 5.27).

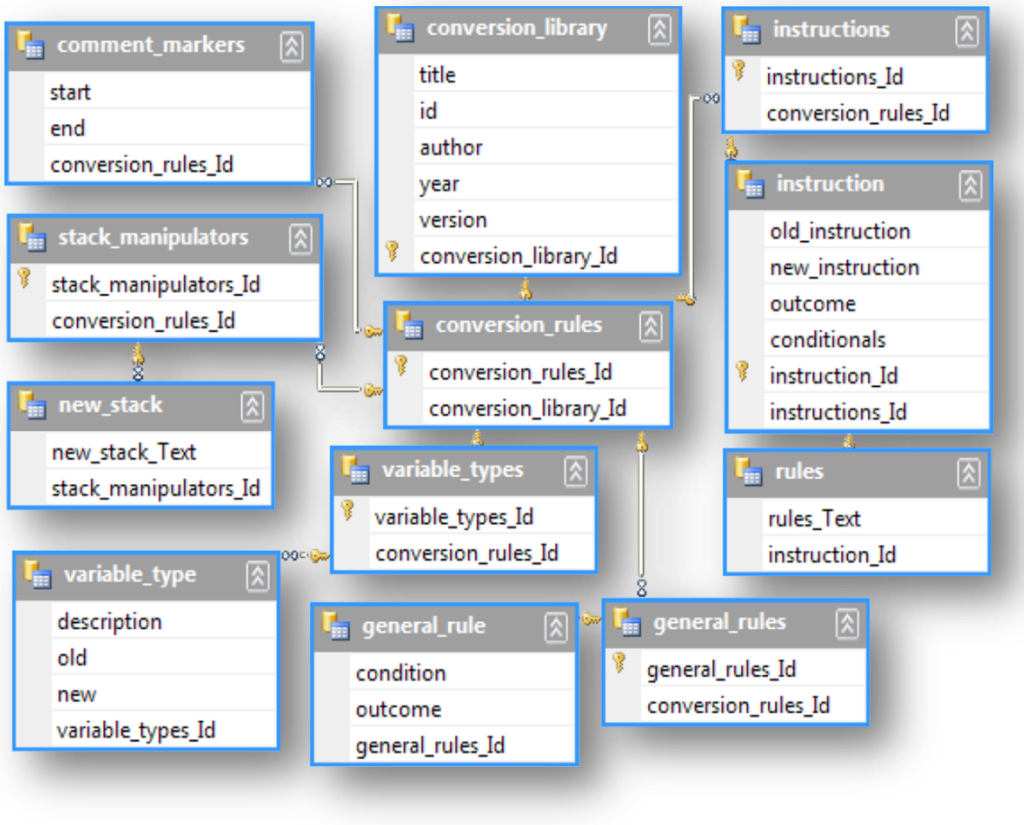

Figura 5.27 – Diagrama de Entidades e Relações do *schema* associado aos dicionários

Segue-se então uma descrição dos seus elementos:

- **conversion_library** – Nó único do XML, como definido pelas especificações XML. Para além do nó **conversion_rules**, contém ainda as informações secundárias como o título do dicionário, autor, ano e versão.
- **conversion_rules** – Nó que contém as regras gerais a aplicar (**general_rules**), os tipos de variáveis suportados (**variable_types**), os marcadores de comentários (**comment_markers**), regras para as instruções (**instructions**) e indicação de quais as instruções que interferem com a pilha de memória.
- **general_rules** – Estas regras são aplicadas independentemente do código, apenas limitadas pelas condicionantes impostas. Podem incluir linhas fixas a acrescentar ao código final ou apagar algo do código gerado. Estas regras são aplicadas antes de o código do programa ser processado, podendo ser utilizadas para vários fins.

- **general_rule** – Elemento das regras gerais.
- **variable_types** – Nó que indica todos os tipos de variáveis suportados. Cada tipo é indicado por uma descrição, o nome do tipo original e o seu correspondente de saída.
- **variable_type** – Elemento dos tipos de variáveis.
- **comment_markers** – Identifica os comentários do código em si, sendo necessários os marcadores de início e fim.
- **instructions** – Para cada instrução que se encontre, há uma regra associada, que inclui o nome da instrução velha e correspondente de saída, condicionantes e resultado de código. Também podem ter um rol de regras a aplicar, que afectem todo o código ou linhas únicas.
- **instruction** – Elemento das instruções.

5.4.3 FUNCIONAMENTO DO *UNIFIL*

O processo de tradução do **UnifIL** recorre às várias classes indicadas no capítulo 4.3.2, cujo funcionamento e implementação são seguidamente explicados. Para fins de clareza, serão detalhados apenas os métodos mais importantes/relevantes.

5.4.3.1 CLASSE INTERFACE

Esta classe (Figura 5.28) tem associados 3 atributos.

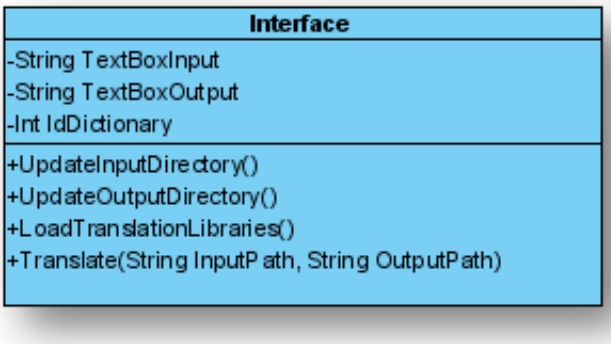

Figura 5.28 - Diagrama de classe do *Interface* do *UnifIL*

Estes definem a chamada do processo de tradução e são introduzidos ou indicados pelo utilizador, nomeadamente:

- Origem do ficheiro a converter (*TextBoxInput*).

- Destino dos ficheiros convertidos (*TextBoxOutput*).
- Dicionário selecionado da lista dos disponíveis (*IdDictionary*).

Os métodos associados a esta classe são:

- *UpdateInputDirectory* – Método que atualiza internamente o ficheiro indicado para tradução. É ativado sempre que há alguma alteração do atributo *TextBoxInput* pelo utilizador, recorrendo para isso à classe *Gestão de Dados*.
- *UpdateOutputDirectory* – Método que atualiza internamente a diretoria de destino do ficheiro traduzido. É ativado sempre que há alguma alteração do atributo *TexBoxOutput* pelo utilizador.
- *LoadTranslationLibraries* – Método que preenche a lista dos dicionários de tradução disponíveis, procurando numa subdiretoria do sistema computacional, através da *Gestão de Dados*.
- *Translate* – Método que chama a tradução por parte da classe *Tradutor*.

5.4.3.1 CLASSE GESTÃO DE DADOS

Esta classe (Figura 5.29) tem associados 2 atributos.

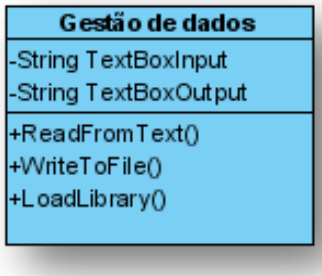

Figura 5.29 - Diagrama de classe da *Gestão de Dados* do *UnifIL*

Estes atributos são introduzidos ou indicados pelo utilizador através do interface, nomeadamente:

- Origem do ficheiro a traduzir (*TextBoxInput*).
- Destino do ficheiro traduzido (*TexBoxOutput*).

Estes 2 atributos definem os caminhos de chamada dos seus métodos. Os métodos associados a esta classe são:

- *ReadFromText* – Método que lê, para memória, o texto integral de um ficheiro a converter.
- *WriteToFile* – Método que escreve o ficheiro de saída a partir da estrutura de dados. O seu funcionamento está explicado mais detalhadamente no capítulo 5.4.6.
- *LoadLibrary* – Método que lê, para memória, o dicionário de regras de tradução escolhido.

5.4.3.2 CLASSE TRADUTOR

Esta classe (Figura 5.30) tem associados 2 atributos.

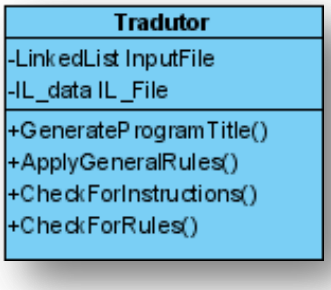

Figura 5.30 - Diagrama de classe do *Tradutor* do *UnifIL*

São esses atributos:

- Ficheiro de entrada em memória (*InputFile*).
- Estrutura de dados em memória, onde é armazenada a tradução antes de ser escrita para ficheiro (*IL_File*).

Os métodos associados a esta classe são:

- *GenerateProgramTitle* – Método que gera o título do programa/POU a partir do nome do ficheiro de entrada.
- *ApplyGeneralRules* – Método que aplica as regras gerais indicadas pelo dicionário de tradução.
- *CheckForInstructions* – Método que recursivamente analisa cada linha de código do ficheiro de entrada, identifica o tipo de linha (declaração de variáveis ou código) e escreve o código de saída, tudo de acordo com as regras indicadas no dicionário de tradução. O seu funcionamento está explicado mais detalhadamente no capítulo 5.4.4.

- *CheckForFules* – Método que aplica as regras associadas a uma dada instrução. O seu funcionamento está explicado mais detalhadamente no capítulo 5.4.4.2.

5.4.3.3 FUNÇÕES/MÉTODOS DE SUPORTE

Os métodos detalhados na descrição de classes da secção prévia (5.4.3.2) representam apenas os mais importantes/relevantes, para fins de clareza. Várias funções/métodos de suporte tiveram que ser desenvolvidos para garantir o bom funcionamento do **UnifIL**, mantendo ao mesmo tempo uma implementação nivelada e hierárquica. Segue-se então uma descrição do papel e funcionamento das funções/métodos de suporte.

- *RemoveComment* – Separa o código útil do texto de comentário de uma dada linha de código, procurando pelo marcadores de comentário indicados pelo dicionário.
- *SeparateInstFromArg* – Função de suporte da análise sintática de código, separa uma *string* pelo primeiro carácter de espaço (" "). Este funcionamento é indicado para a separação de instrução e argumento das várias linguagens de CLP proprietárias.
- *ReplaceIllegalChars* – Carateres especiais, como tabulações e quebras de linha, complicam a tarefa da análise sintática, sendo esta função utilizada para a substituição de carateres disruptivos do processo de análise sintática, por um carácter de espaço.
- *IsFileEmpty* – Verifica se o ficheiro indicado possui algum texto ou se se encontra vazio. Utilizado para negar o processo de conversão, caso o ficheiro não contenha dados.
- *GetFileFromAssembly* – São utilizados alguns elementos no sistema computacional, como imagens do interface e o *schema* XML de validação dos dicionários (ver capítulo 5.4.2), que estão embutidos sob a forma de *assemblies*, dados embutidos no binário da aplicação.
- *ValidateAssemblies* – As *assemblies* são suscetíveis de corrupção, o que poderá originar um resultado de conversão incoerente com o previsto. Para evitar tais situações, todas as *assemblies* são verificadas através de um processo de *hashing* e comparação. Isto garante, probabilisticamente, a integridade de todas as *assemblies*. Esta verificação ocorre cada vez que o sistema computacional é iniciado.
- *CompareWithIgnores* – Função que informa sobre a coerência de duas *strings*, ignorando carateres especiais. É usada para comparar variáveis com os tipos definidos no dicionário.

- *GetIndexFromOrders* – Função que devolve a posição de uma instrução de manipulação de pilha que cumpras os requisitos dados. Utilizada para traduções de operações de pilha.
- *GetXmlNodeValueByTag* – Função para ler o valor de um nó XML com apenas o nome como referência.

5.4.3.4 SEQUÊNCIA DE TRADUÇÃO

O diagrama de sequência Figura 5.31 ilustra o funcionamento interno do **UnifIL** mais detalhadamente, aquando de uma tradução.

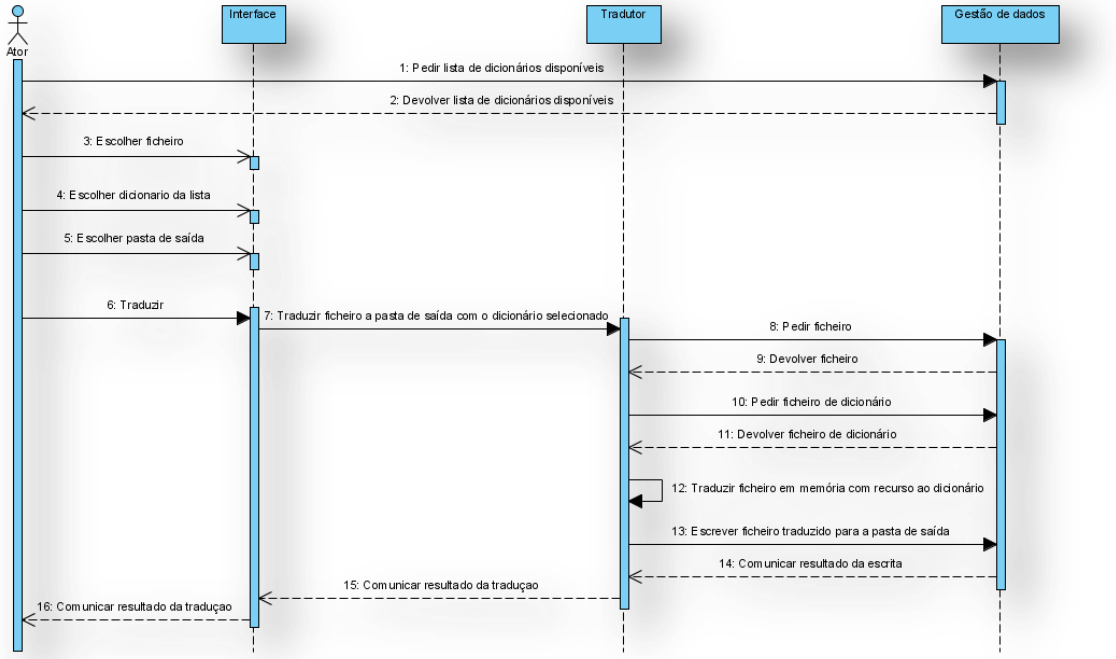

Figura 5.31 - Diagrama de sequência da tradução de código CLP proprietário para IL

Segue-se uma descrição dos vários passos desta tradução:

1. Ao ser inicializado, o **UnifIL** pede a lista dos dicionários disponíveis à classe *Gestão de Dados*.
2. Obtém essa lista e disponibiliza-a no interface.
3. Através da *Interface*, o utilizador seleciona qual o ficheiro a converter.
4. O destino do ficheiro traduzido é gerado automaticamente a partir da origem, mas pode ser modificado pelo utilizador.

5. O utilizador indica, da lista dos dicionários de tradução disponíveis, qual a utilizar.
6. O utilizador aciona a tradução do ficheiro selecionado.
7. Acionar a tradução na classe *Tradutor*.
8. O *Tradutor* pede o ficheiro indicado à *Gestão de Dados*.
9. A *Gestão de Dados* devolve ao *Tradutor* o ficheiro pretendido sob a forma de uma listagem de texto em memória.
10. O *Tradutor* pede o dicionário indicado à *Gestão de Dados*.
11. A *Gestão de Dados* devolve ao *Tradutor* o dicionário pretendido sob a forma de uma listagem de texto em memória.
12. O processo de tradução é ativado. Este está detalhadamente explicado no próximo capítulo (5.4.4).
13. É escrito para ficheiro o resultado da tradução, a partir da estrutura de dados em memória, gerada pelo processo de tradução. O processo de escrita para ficheiro está detalhadamente explicado no capítulo 5.4.6.
14. Resposta de sucesso na escrita do ficheiro.
15. Resposta de sucesso de tradução completa.
16. O *Interface* informa o utilizador, através de uma caixa de mensagem, do sucesso da tradução.

De notar que qualquer erro na tradução interromperá imediatamente o processo de tradução, mostrando a mensagem de erro apropriada.

5.4.4 PROCESSO DE TRADUÇÃO

O processo de tradução do ficheiro está dividido em 3 fases distintas (como ilustrado no diagrama de atividades da Figura 5.32), através de 3 métodos, como indicados na descrição da classe *Tradutor* (capítulo 5.4.3.2).

Os 3 métodos são chamados ordenadamente e vão preenchendo a estrutura de dados com os seus resultados. São eles:

- *GenerateProgramTitle* (Gerar título) – Para a fase de geração do nome, dado que os códigos proprietários variam na inclusão de título ou nome de programa, o tradutor assume sempre, como título, o nome do ficheiro a traduzir. Este nome é copiado para a estrutura de dados.
- *ApplyGeneralRules* (Aplicação de regras gerais).
- *CheckForInstructions* (Traduzir código).

Estes 2 últimos métodos são explicados em maior detalhe nos próximos capítulos.

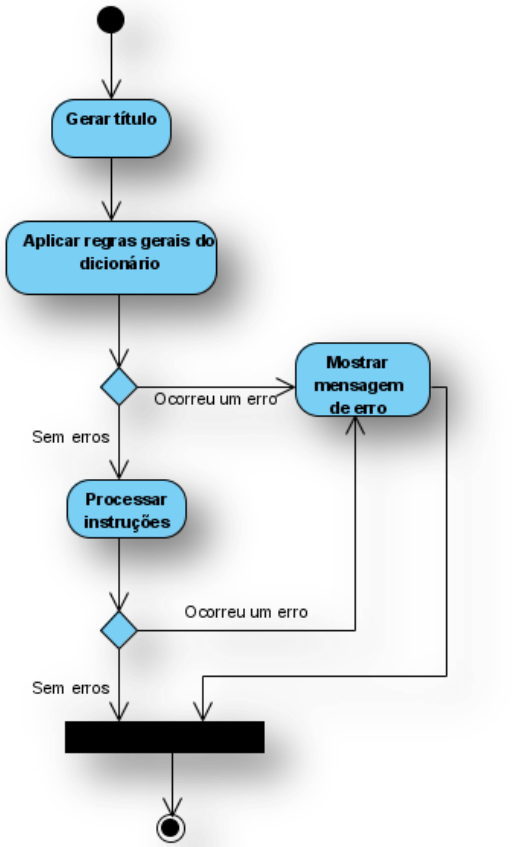

Figura 5.32 - As 3 fases do processo de tradução

5.4.4.1 APPLYGENERALRULES – APLICAÇÃO DE REGRAS GERAIS

As regras gerais a aplicar são um suporte de adição de código e/ou comentário ao ficheiro de saída, incondicionalmente. Isto é útil para linguagens proprietárias com algum requisito funcional específico ou simplesmente para acrescentar notas e comentários associados ao dicionário de tradução. As regras são lidas do dicionário e acrescentadas às primeiras linhas de código da estrutura de dados. O código fonte da implementação desta classe está disponível para consulta no Anexo X.

5.4.4.2 CHECKFORINSTRUCTIONS – TRADUZIR CÓDIGO

O processo de tradução é executado através uma análise sintática (*parsing*) linha a linha de acordo com os valores fornecidos pelo dicionário de tradução. O código fonte da implementação

desta classe está disponível para consulta no Anexo XI. O comportamento do método *CheckForInstructions* depende da instrução encontrada e as regras associadas a essa instrução, assim como a variável de argumento e sua possível declaração. O diagrama de atividades deste comportamento está ilustrado na Figura 5.33. Como se pode observar, todas as operações dependem das informações fornecidas pelo dicionário de tradução e qualquer erro que ocorra interromperá todo o processo. Este processo ocorre para todas as linhas de código, até ao fim do ficheiro de entrada. Será portanto explicado referindo-se sempre à atual linha em análise.

Separando as atividades em fases, fica-se com:

- Fase de procura – consiste em comparar a instrução na linha atual com todas as instruções definidas pelo dicionário. Caso uma instrução compatível seja encontrada, o processo pode prosseguir.
- Fase de declaração – averigua-se se a variável do argumento já existe na estrutura de dados. Caso não exista, é identificado o seu tipo, comparando com os tipos definidos no dicionário e declarada como o tipo compatível encontrado.
- Fase de aplicação – a partir das indicações simbólicas, associadas à instrução vigente, é gerada o código de saída e escrito para a estrutura de dados.
- Fase de regras – caso a instrução vigente tenha regras extra associadas, estas são aplicadas.

Estas duas últimas fases são cruciais no método *CheckForInstructions*, pelo que se expande o seu funcionamento nos capítulos seguintes.

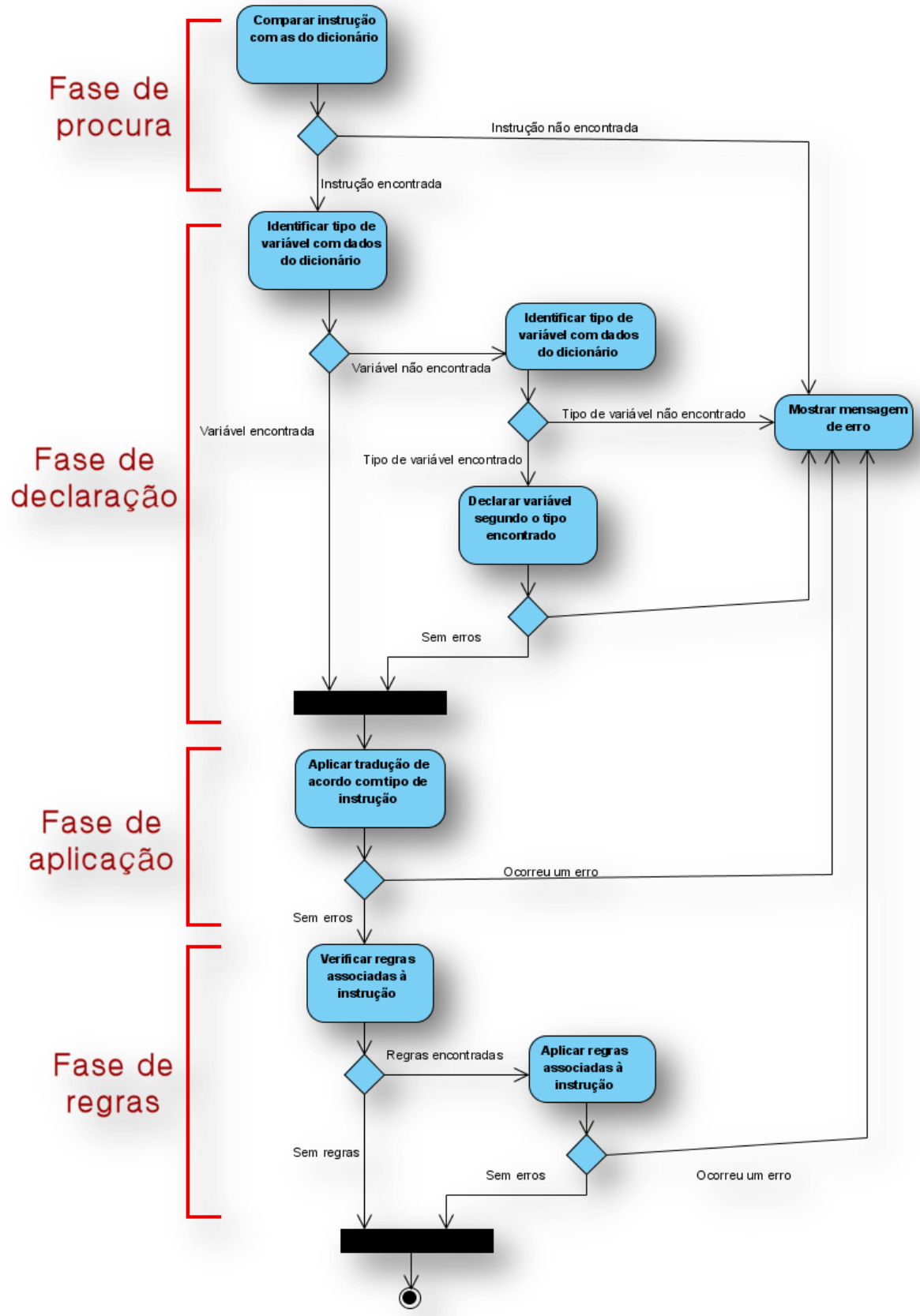

Figura 5.33 - Diagrama de atividades da tradução de uma linha

5.4.4.1 APPLYOUTPUTLINE – APLICAR LINHA TRADUZIDA

Este método escreve, para a estrutura de dados, a linha de código traduzida, como indicada o campo "outcome" de cada instrução indicada no dicionário de tradução. Este campo especifica, como código de saída, texto literal e/ou valores simbólicos, como o argumento original ou novas instruções. Os valores simbólicos típicos são:

- "*new_Instruction*" – Refere-se à instrução indicada pelo campo da nova instrução a ser utilizada.
- "*old_instruction*" – Refere-se à instrução original da linha a traduzir.
- "*argument*" – Refere-se argumento original da linha a traduzir.

O uso de literais, pode complementar algum requisito específico da linguagem proprietária em questão. Mas para casos de resolução mais complexa, existem as regras associadas à instrução.

5.4.4.2 CHECKFORFULES – APLICAR REGRAS ASSOCIADAS A INSTRUÇÃO

Certos eventos são ativados por regras associadas às instruções, sendo o caso mais recorrente o da declaração de variáveis. Várias linguagens proprietárias não incluem a declaração de variáveis com elemento formal do código, mas sim como campos introduzidos separadamente no IDE proprietário. Não tendo acesso a essas declarações (podendo nem existir em alguns casos), o **UnifIL** terá que induzir as variáveis e seu tipo puramente a partir do código de programa. Isto é efetuado com recurso às indicações relevantes no dicionário de tradução, utilizando uma comparação especial (ver capítulo 5.4.3.3) para identificar os tipos de variáveis subjacentes. Para declarar uma variável, é necessário saber o seu tipo de dados e *I*/*O*, sendo ambos inferidos pela sua nomenclatura e referência correspondente no dicionário de tradução dos elementos do tipo "*variable_type*". Esta inferência é feita através do método *CompareWithIgnores*, referido no capítulo 5.4.3.3. O asterisco indica texto variável, aceitando portanto variáveis que sejam da forma "I0.1", "I1.0", "I10.2", (…) serão aceites como formas de "I*.*".

Outras regras são também aplicadas deste modo, como é o caso de operações de pilha de acumuladores, explicadas no capítulo 5.1.4. O código fonte da implementação desta classe está disponível para consulta no Anexo XII.

5.4.5 RETENÇÃO DE COMENTÁRIOS

Dado que o objetivo principal do presente trabalho é possibilitar testes simulados para estudo, fez-se questão de manter intactos quaisquer comentários que estivessem no código de programa original, transplantando-os para o código IL de saída. Como cada linguagem proprietária de CLP possui um conjunto de marcadores, normalmente um carácter ou um conjunto de carateres, que identificam o começo e fim dos comentários de cada linha, estes deverão ser indicados no dicionário de tradução, como no exemplo da Figura 5.34.

```
<comment_markers>
<start>//</start>
<end>EOL</end>
</comment_markers>
```

Figura 5.34 - Exemplo de marcadores de comentários num dicionário

Este exemplo mostra os marcadores de comentários que indicam, como início, o conjunto de carateres "//" e, como fim, "EOL" que significa o fim da linha (*End Of Line*).

Os comentários de cada linha do código original são copiados para a estrutura de dados e apagados antes da análise sintática processar a linha. Aquando da escrita para o ficheiro IL de saída, os comentários serão copiados da estrutura de dados para linha da instrução associada. Isto ajudará a localizar marcadores ou códigos problemáticos do código original, identificando-os no código traduzido.

5.4.6 ESCRITA PARA FICHEIRO DE IL – *WriteToFile*

O método de escrita para ficheiro, da classe *Gestão de Dados*, utiliza as informações na estrutura de dados para gerar o ficheiro de saída em código em IL, segundo o modelo representado na Figura 5.35.

De acordo com o tipo de I/O (entrada, saída, global, interna) das variáveis, estas serão declaradas no campo adequado, juntamente com o seu tipo de dados, valor inicial e campo de retenção. O código de programa é copiado para a zona adequada, onde os comentários de cada linha são inseridos com os marcadores de comentário do IL. Finalizando o código com a linha de fim de programa ("END_PROGRAM"), o ficheiro é copiado da memória para o destino selecionado, com o nome indicado na estrutura de dados.

```
PROGRAM Nome_do_POU (*Comentário do POU*)

VAR_INPUT
(…)
(Declaração de variáveis de entrada)
(…)
END_VAR
VAR_OUTPUT
(…)
(Declaração de variáveis de saída)
(…)
END_VAR

(…)
(Código de programa)
(…)

END_PROGRAM
```

Figura 5.35 - Código modelo de IL

5.4.7 EXEMPLO ILUSTRATIVO DE TRADUÇÃO

Aquando da inicialização do tradutor, este procura ficheiros XML na subdiretoria "*Dictionaries*", de onde o tradutor está a correr. Escolhidos os parâmetros adequados, tomemos um exemplo de código simples, em linguagem STL-200 da *Siemens* para a tradução:

```
LD   I0.0//start program
A    I0.1
O    I0.2
=    Q0.0
```

Figura 5.36 - Exemplo de código STL a traduzir

Pelo diagrama da Figura 5.32, o 1º passo será o da geração do título. O tradutor assume sempre, como título, o nome do ficheiro a traduzir. Neste exemplo, o nome do ficheiro é "STL test code.stl", ficando o programa de CLP então com o título "STL_test_code" – os carateres "ilegais" em IL normalizado são substituídos por um traço inferior (_) por motivos de compatibilidade.

Os troços, relevantes para esta tradução exemplo, do dicionário de tradução correspondente, estão ilustrados na Figura 5.37.

Tendo estes elementos para consulta, pode-se prosseguir ao 2º passo, que será a aplicação de regras gerais. Consultando o elemento de regras gerais ("*general_rule*"), obtém-se apenas uma linha de comentário a adicionar no início do código traduzido.

As aspas indicam que se trata de texto literal a incluir no código de saída e o comentário é indicado pelos carateres "//". No entanto, o tradutor não sabe isto à partida, referenciando para isso o campo dos marcadores de comentários desta linguagem ("*comment_markers*").

```
<general_rule>
<condition></condition>
<outcome>"//Siemens Step 7 S-200 STL translation dictionary"</outcome>
</general_rule>
<general_rule>
<condition></condition>
<outcome>""</outcome>
</general_rule>

<comment_markers>
<start>//</start>
<end>EOL</end>
</comment_markers>

<variable_type>
<description>digital input</description>
<old>I*.*</old>
<new>input digital</new>
</variable_type>

<variable_type>
<description>digital output</description>
<old>Q*.*</old>
<new>output digital</new>
</variable_type>

<instruction>
<old_instruction>=</old_instruction>
<new_instruction>ST</new_instruction>
<rules>declare argument</rules>
<outcome>instruction argument</outcome>
<conditionals></conditionals>
</instruction>

<instruction>
<old_instruction>A</old_instruction>
<new_instruction>AND</new_instruction>
<rules>declare argument</rules>
<outcome>instruction argument</outcome>
<conditionals></conditionals>
</instruction>

<instruction>
<old_instruction>O</old_instruction>
<new_instruction>OR</new_instruction>
<rules>declare argument</rules>
<outcome>instruction argument</outcome>
<conditionals></conditionals>
</instruction>

<instruction>
<old_instruction>LD</old_instruction>
<new_instruction>LD</new_instruction>
<rules>declare argument</rules>
<outcome>instruction argument</outcome>
<conditionals></conditionals>
</instruction>
```

Figura 5.37 - Troços aplicáveis ao exemplo do dicionário de tradução correspondente

O marcador de início de comentário será "//" e marcador de fim é o fim da linha (EOL, do inglês *End Of Line*), pelo que o tradutor assume, como comentário, todo o texto após o marcador de início. Esta linha de comentário é então adicionada ao código de saída.

Findas as regras gerais, o tradutor corre todas as linhas restantes de programa, comparando as instruções com as do dicionário ("*instruction*").

A linha 1 tem como instrução "LD", cujo código de saída, associado está no campo "*outcome*". Este especifica como código de saída, os valores simbólicos de "*instruction*" e "*argument*" que correspondem à nova instrução (campo "*new_instruction*") e ao argumento original do código,

ficando a saída como "LD I0_0" – os carateres "ilegais" em IL normalizado são substituídos. Esta linha é adicionada à estrutura de dados. Aplicando as regras associadas a esta instrução (campo "*rules*"), diz-nos para declarar o argumento utilizado.

Correspondendo ao argumento da linha em análise ("I0.5"), é verificada na estrutura de dados se já foi previamente declarada e, caso negativo, é declarada com as informações dadas pelo campo "*new*" que indica "*input digital*", portanto, uma variável booleana do tipo de entrada. Este procedimento de declaração de variáveis é análogo para todos os tipos.

Na linha 2, a instrução é "A", cujo elemento de dicionário correspondente indica como código de saída, os valores simbólicos de "*instruction*" e "*argument*" que correspondem à nova instrução (campo "*new_instruction*") e ao argumento original do código, ficando a saída como "AND I0_1". Esta linha é adicionada à estrutura de dados.

As linhas 3 e 4 são análogas, apenas com a instrução diferente ("O" e "="). Como resultado final, teremos o código da Figura 5.38.

```
PROGRAM EXAMPLE_PROGRAM

VAR_INPUT
I0.0 :BOOL;
I0.1:BOOL;
I0.2:BOOL;
END_VAR
VAR_OUTPUT
Q0.0:BOOL;
END_VAR

(*Siemens Step 7 S-200 STL translation dictionary*)

LD I0_0 (*START PROGRAM*)
AND I0_1
OR I0_2
ST Q0_0

END_PROGRAM
```

Figura 5.38 - Resultado exemplo ilustrativo de tradução

6 EXPERIÊNCIAS DE VALIDAÇÃO

Para validar o trabalho, foram efetuados os processos de tradução e de conversão, com recurso aos sistemas computacionais desenvolvidos, o **UnifIL** e o **Matlaber**. Os testes consistiram em 3 experiências:

1. Traduções de dois programas funcionalmente idênticos, mas em linguagens proprietárias diferentes, para IL normalizado, recorrendo ao **UnifIL**.
2. Conversão de um programa de controlo em IL normalizado para *Matlab*, recorrendo ao **Matlaber**.
3. Combinação da tradução de um programa de controlo, em linguagem proprietária, para IL normalizado mais a conversão para *Matlab*, recorrendo ao **UnifIL** e ao **Matlaber**, respetivamente.

No teste 1, foi criado um programa de testes abstrato para comparação entre as duas linguagens. Nos teste 2 e 3, os programas de controlo resultantes foram aplicados numa simulação para validação do seu funcionamento.

6.1 EXPERIÊNCIAS DE TRADUÇÃO

Para os testes de tradução escreveram-se 2 programas de controlo de CLP funcionalmente idênticos, cada um numa linguagem proprietária diferente, como se pode observar na Figura 6.1.

Código em Q Series

```
1  ;start program
2  LD    I0.1
3  CJ    P0
4  LD    I0.5
5  AND   I0.3
6  LD    M0.0
7  ORI   M0.1
8  ORB       ; OR block
9  OUT   M0.0
10 JMP   P1
11 P0
12 LD    0
13 OUT     M0.0
14 P1
```

Código em STL

```
1  //start program
2  LD    I0.1
3  JMP    0
4  LD    I0.5
5  A     I0.3
6  LD    M0.0
7  ON    M0.1
8  OLD   //OR block
9  =     M0.0
10 LD    1
11 JMP   1
12 LBL 0
13 LD 0
14 =     M0.0
15 LBL 1
```

Figura 6.1 - As 2 versões proprietárias do código do programa de testes

As linguagens de programação proprietárias utilizadas foram:

- *Q Series* (modo *List Mode*, semelhante ao IL) da *Mitsubishi*.
- *S200* (código STL, semelhante ao IL) da *Siemens*.

Foram desenvolvidos, para o trabalho apresentado, dicionários de regras de tradução (ver capítulo 5.4.2) para ambas as linguagens apresentadas. O funcionamento do programa de teste, a ser traduzido, é abstrato e serve apenas como demonstração das capacidades de tradução, recorrendo a típicas instruções e operações de CLP, como lógica booleana, assim como algumas operações de fluxo de programa, como *labels* e *jumps*.

Traduzindo estes códigos com recurso ao **UnifIL** (como ilustrado na Figura 6.2), para IL normalizado, espera-se obter um resultado sob a forma de um programa funcionalmente idêntico de ambas as traduções.

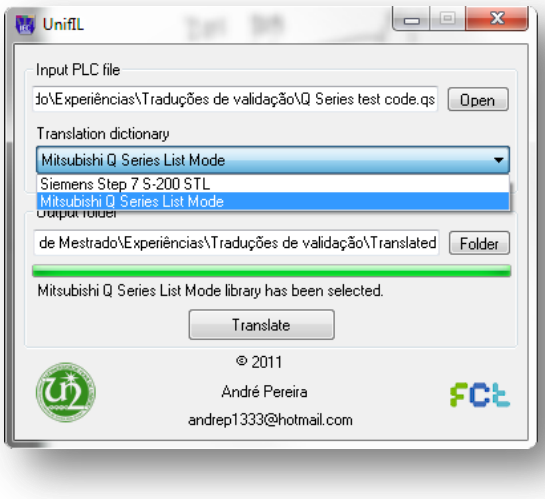

Figura 6.2 - Interface do *UnifIL* aquando da seleção do dicionário de tradução adequado

Como se pode constatar na Figura 6.1, existem diferenças substanciais entre os dois códigos. Possuem instruções semelhantes, que diferem apenas no nome do operador: como exemplo, a disjunção lógica negada (*Not Or*) é "ORI" em *Q Series* e "ON" em STL. Os marcadores de comentários (";" e "//") também são diferentes. Instruções de fluxo já diferem substancialmente, pois o STL apenas possui *jumps* condicionais, não possuindo um *jump* **in**condicional como o *Q Series* possuí ("JMP" – *jump*). Tais limitações tiveram que ser levadas em conta na escrita do programa de testes em cada linguagem.

Estas diferenças entre linguagens apenas servem para ilustrar que não basta substituir palavras para as linguagens se tornarem equivalentes, sendo necessário um ajuste dinâmico ao código apresentado. Aquando de uma tradução estas *nuances* são todas processadas no tradutor através do dicionário de tradução. Para traduzir ambos os códigos, recorrendo ao **UnifIL**, basta escolher o código de origem e o dicionário de tradução adequado (como ilustrado na Figura 6.2).

Efetuada a tradução, obtiveram-se os seguintes troços funcionais de código (Figura 6.3). Estes códigos não incluem cabeçalho nem declaração de variáveis, para fins de clareza. A listagem completa do código traduzido pode ser consultada nos Anexos XIII e XIV.

Tradução de *Q Series*

```
19  (* START PROGRAM *)
20  LD I0_1
21  JMPC P0
22  LD I0_5
23  AND I0_3
24  OR( M00
25  ORN M01
26  ) (*  OR BLOCK *)
27  ST M00
28  JMP P1
29  P0:
30  LD 0
31  ST M00
32  P1:
33
34  END_PROGRAM
```

Tradução de STL

```
19  (*START PROGRAM *)
20  LD I0_1
21  JMPC P0
22  LD I0_5
23  AND I0_3
24  OR( M00
25  ORN M01
26  ) (*  OR BLOCK *)
27  ST M0_0
28  LD 1
29  JMPC 1
30  0:
31  LD 0
32  ST M0_0
33  1:
34  END_PROGRAM
```

Figura 6.3 - Troços funcionais do código traduzido

Pode-se então comparar os resultados que, deveriam ser funcionalmente, idênticos. Uma análise rápida mostra que os resultados da tradução são praticamente idênticos, tirando o formato dos nomes das *labels* e a linha 29 da tradução de STL. Analisando mais detalhadamente o ponto crucial, a diferença da linha 29 não é nada menos que a expressão da limitação já referida do STL apenas possuir *jumps* condicionais. No entanto, a funcionalidade retém-se a 100%.

Pode-se então concluir a eficácia do tradutor, ao normalizar 2 programas em linguagens proprietárias diferentes em IL funcionalmente idêntico.

6.2 EXPERIÊNCIAS DE SIMULAÇÃO

Para comprovar e validar o funcionamento do trabalho desenvolvido, foram testados 2 processos, já modelados em *Matlab*, que foram controlados pelo código traduzido e/ou convertido. O método de teste consiste na tradução e/ou conversão do código do programa de controlo para o ficheiro ".m", sendo este depois integrado no ambiente de simulação *Matlab*/*Simulink*, controlando o modelo simulado do processo industrial em questão.

6.2.1 SIMULAÇÃO 1: TANQUE

Para 1ª experiência, foi utilizado um modelo de um sistema linear pré-existente. Consiste num tanque de água cujo nível tem que ser mantido entre os 4 e 5 metros de altura (Figura 6.4), sendo o fluxo de entrada de água limitadamente aleatório.

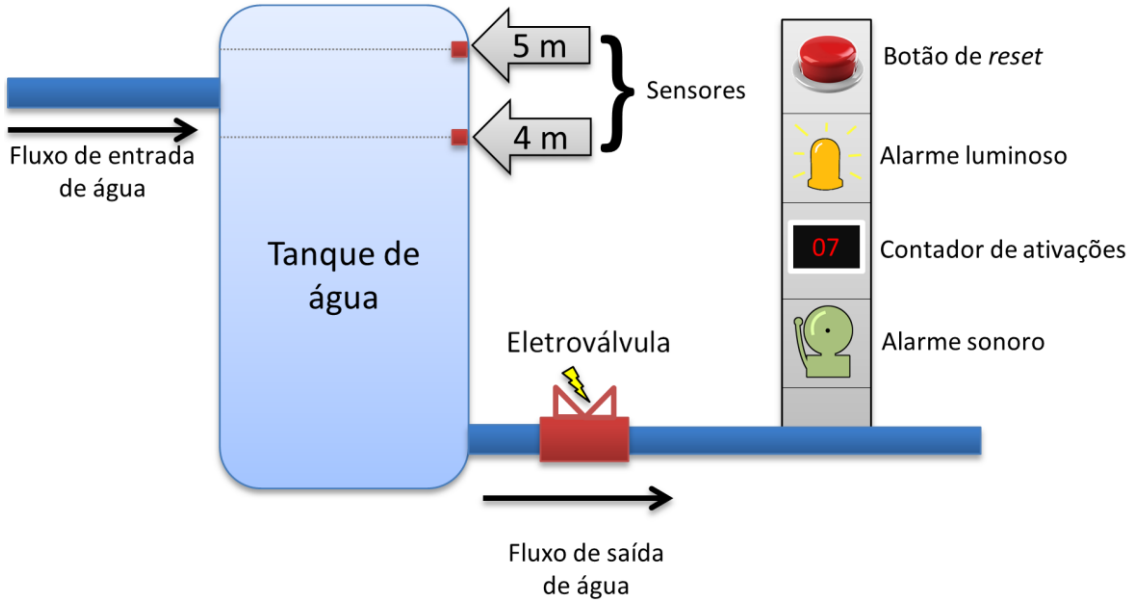

Figura 6.4 - Representação do modelo de tanque (simulação 1)

Para controlar o nível de água, existe uma electroválvula, comandada por solenoide, para escoamento, controlada por CLP. O CLP controla esta válvula de acordo com os dados obtidos por dois sensores de nível, instalados a 4 e 5 metros de altura, respetivamente. O fluxo de água através da válvula é fixo e maior que o máximo fluxo de entrada. O controlo do nível água a implementar e o seu ciclo de funcionamento podem ser descritos pelo SFC da Figura 6.5. Este controlo apenas ativa ou desativa a válvula quando estritamente necessário, evitando desgaste do equipamento.

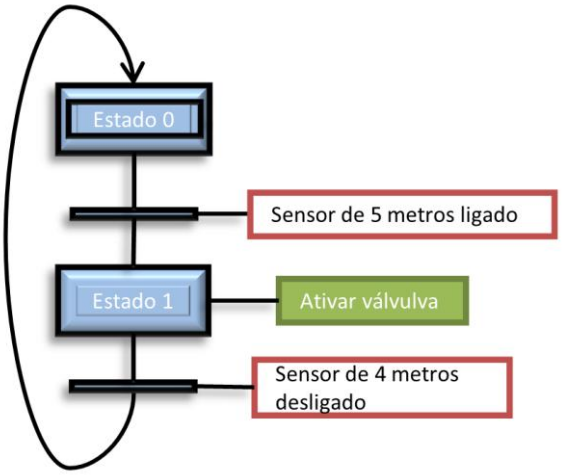

Figura 6.5 - Diagrama funcional (SFC) do controlo de nível da simulação 1

Para fins de manutenção, requer-se também uma contagem do número de vezes que a válvula é ativada, de forma a monitorar o seu desgaste e prevenir falhas. Quando esta contagem atingir as 8 vezes, deverão soar 2 tipos de alarmes: luminoso e sonoro. O alarme luminoso deverá disparar assim que a contagem de ativações chegar aos 8 valores. O alarme sonoro só deverá disparar 1 segundo após o luminoso ter disparado, mantendo-se ligado até que um *reset* externo limpe o valor do contador e desligue ambos os alarmes, através de um botão no interface. O funcionamento deste controlo, pode ser consultado no SFC da Figura 6.6. Para efeitos de teste, este botão de *reset* será pressionado aos 3 e 12 segundos.

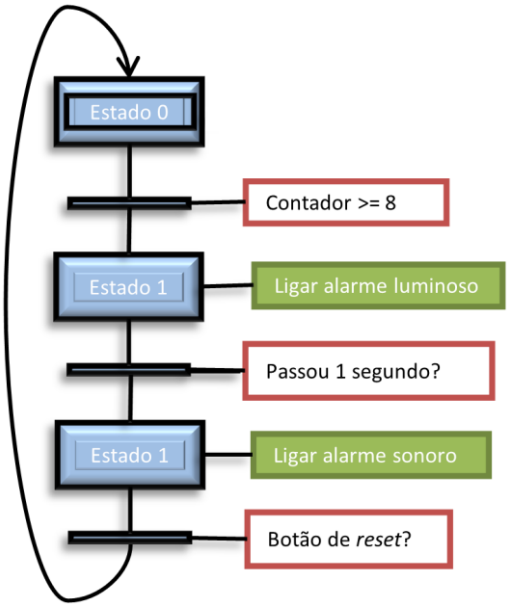

Figura 6.6 - Diagrama funcional (SFC) dos alarmes da simulação 1

Implementando este controlo sob a forma de um programa de CLP escrito em IL, obteve-se o troço funcional do seu código que pode ser consultado na Figura 6.7. A listagem completa pode ser consultada no Anexo XV.

```
LD valve
JMPC VALVEON
(*Valve not on branch*)
LD Maximum_Level_Switch;
JMPCN ALLDONE
(*If maximum level is true*)
S valve;
S Counter1.CU;
JMP ALLDONE
(*If Valve on branch*)
VALVEON:
LD Minimum_Level_Switch;
JMPC ALLDONE;
(*If water is below minimum and valve is on*)
R valve;
R Counter1.CU;
ALLDONE: (*valve control calculated, onto alarm program*)
CAL
Counter1(Reset:=External_Counter_Reset,PV:=8,CV=>alarm_counter)

CAL Timer1(IN:=Counter1.Q,PT:=1,ET=>0)

LD Timer1.Q
ST Buzz_Signal_Alarm;

LD Counter1.Q;
ST Light_Signal_Alarm;
```

Figura 6.7 - Código funcional do programa de controlo da simulação 1

Criando a montagem, em ambiente *Simulink*, do modelo em estudo previamente modelado, obteve-se a representação da Figura 6.8. O bloco "*Water Tank System*" encontra-se já modelado e representa o tanque de água, sensores e válvula. A representação segue o modelo generalista indicado no capítulo 5.1.3. Alterações de nota incluem os blocos: entrada de água ("*Water in*"), o gerador de relógio ("*Clock*"), os sinais do botão de interface ("*External Counter Reset*") e vários mostradores.

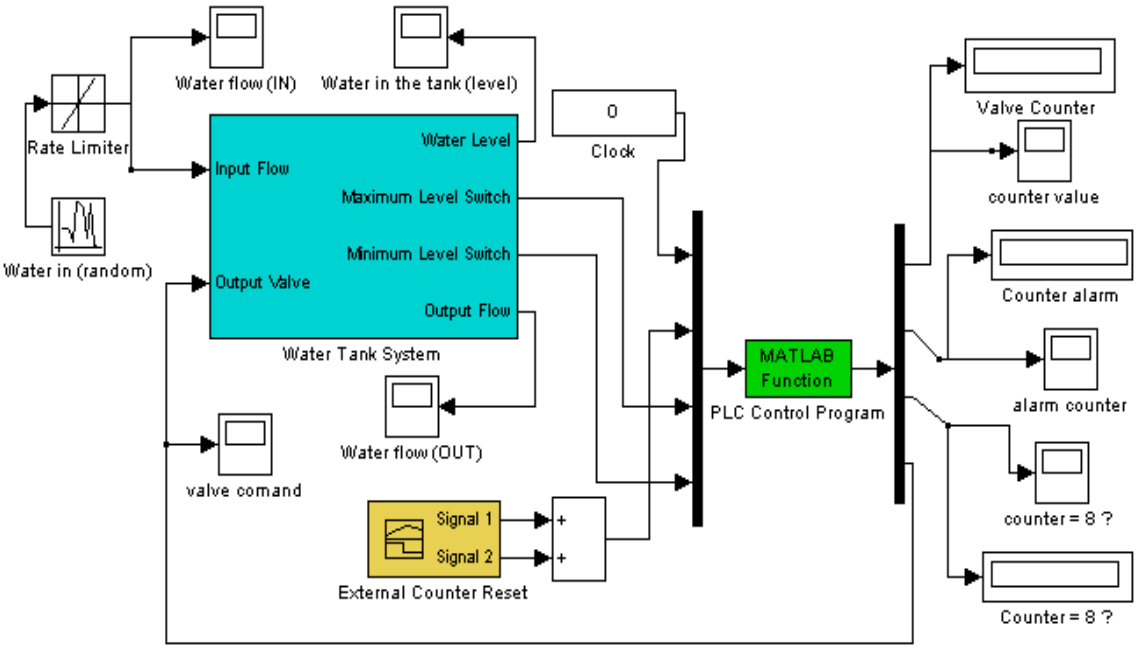

Figura 6.8 - Representação em *Simulink* do modelo de tanque (simulação 1)

É então necessário modelar o programa de controlo da Figura 6.7 para a forma proposta pelo presente trabalho. Usando a conversão para ficheiro ".m" com recurso ao sistema computacional desenvolvido, o **Matlaber**, obteve-se o troço funcional do código ilustrado na Figura 6.9. A listagem completa deste código pode ser consultada no Anexo XVI.

Uma breve análise do código convertido, revela os efeitos aparentes do fluxo de programa por *jumps*, tendo gerado uma variedade de *labels*, *gotos* e *returns*. O funcionamento vetorial da chamada dos POU também é visível. Este código será chamado recursivamente pelo bloco "*MATLAB Function*" da simulação (Figura 6.8).

```
if (VALVE==1)
    goto('VALVEON');
    return % Part of the goto functionality in Matlab, ignore
end
% VALVE NOT ON BRANCH ||
if (MAXIMUM_LEVEL_SWITCH==0)
    goto('ALLDONE');
    return % Part of the goto functionality in Matlab, ignore
end
% IF MAXIMUM LEVEL IS TRUE ||
VALVE = 1;
COUNTER1(2) = 1; % COUNTER1.CU
goto('ALLDONE');
return % Part of the goto functionality in Matlab, ignore
% Part of the goto functionality in Matlab, ignoreIF VALVE ON BRANCH
||
% LABEL VALVEON
if (MINIMUM_LEVEL_SWITCH==1)
    goto('ALLDONE');
    return % Part of the goto functionality in Matlab, ignore
end
% IF WATER IS BELOW MINIMUM AND VALVE IS ON ||
VALVE = 0;
COUNTER1(2) = 0; % COUNTER1.CU
% ALLDONE LABEL'S COMMENT:
% LABEL ALLDONE
COUNTER1(5)=EXTERNAL_COUNTER_RESET;COUNTER1(4)=8;
COUNTER1=COUNTER_UP(COUNTER1);ALARM_COUNTER=COUNTER1(3);
TIMER1(2)=COUNTER1(6);TIMER1(4)=1;TIMER1=TIMER_ON(CLOCK_IN,TIMER1);
%COUNTER1.Q
BUZZ_SIGNAL_ALARM = TIMER1(6); % COUNTER1.QTIMER1.Q
LIGHT_SIGNAL_ALARM = COUNTER1(6); % COUNTER1.Q
```

Figura 6.9 – Código funcional de controlo da simulação 1 convertido para *Matlab*

As correspondências das entradas e saídas do CLP de controlo com a descrição do seu papel podem ser consultadas na Tabela 6.1.

Os endereços são indicados por variáveis às quais foram atribuídos portos de entrada e saída no IDE de programação de CLP, tornando a programação agnóstica relativamente aos endereços de I/O.

Tabela 6.1 – Descrição dos sinais, ações e endereços de memória do CLP de controlo do tanque

Descrição do sinal	Descrição da ação	Endereço I/O no CLP	Variável correspondente em *Matlab/Simulink*
Sensor de 5 metros	Indica quando a água está nos 5 m ou acima	Maximum_Level_Switch	MAXIMUM_LEVEL_SWITCH
Sensor de 4 metros	Indica quando a água está nos 4 m ou acima	Minimum_Level_Switch	MINIMUM_LEVEL_SWITCH
Botão de interface	Limpar contagem de ativações da válvula	External_counter_reset	EXTERNAL_COUNTER_RESET
Válvula elétrica	Abre o fluxo de saída do tanque	Valve	VALVE
Alarme luminoso	Sinaliza a contagem de 8 ativações da válvula	Light_Signal_Alarm	LIGHT_SIGNAL_ALARM
Alarme sonoro	Ativado 1 segundo depois do alarme luminoso	Buzz_Signal_Alarm	BUZZ_SIGNAL_ALARM
Contador	Contador de ativações da válvula	Counter1	COUNTER1
Temporizador	Temporizador do alarme sonoro	Timer1	TIMER1

Correndo a simulação, obtiveram-se os resultados de sinais de controlo, contadores e sensores indicados na Figura 6.10. Como é possível constatar, o controlo de nível cumpre os requisitos completamente, ao manter o nível de água entre os 4 e 5 m, apenas alternando entre aberto/fechado quando o nível de água atinge um dos limites. O contador de ativações é incrementado cada vez que a torneira é ativada e o alarme luminoso é ativado sempre que este contador chega aos 8 valores. O alarme sonoro apresenta um atraso de um segundo relativamente ao alarme luminoso, visível por volta dos 8 segundos.

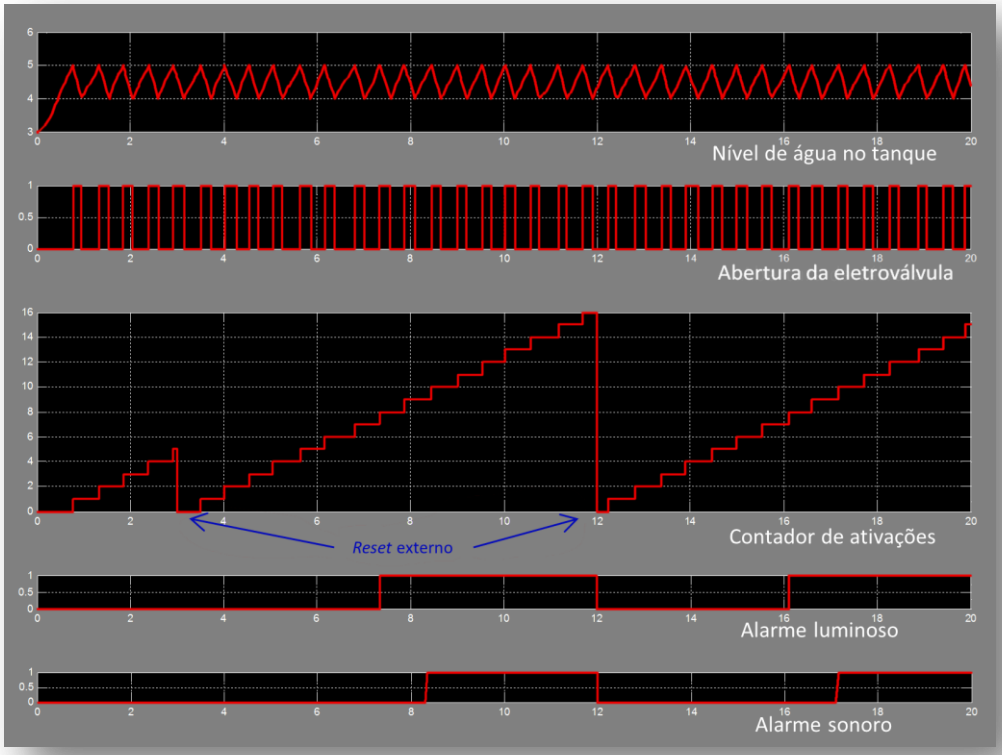

Figura 6.10 - Resultados da simulação da simulação 1

Como anteriormente indicado, aos 3 e 12 segundos é limpo o valor do contador, através do *reset* externo, como indicado na Figura 6.10. Aos 3 segundos, como nenhum dos alarmes está ativado, o *reset* não afeta nada. Mas aos 12 segundos, ambos os alarmes são desativados pelo sinal de *reset*.

Pelos resultados da simulação, o programa de controlo funciona perfeitamente, cumprindo todos os requisitos a que se propõe. A conversão resultou num código limpo, legível por humanos e imediatamente mais aparente que o original em IL, devido ao formato da conversão simbólica, retendo no entanto o fluxo de programa original.

Por esta análise de conversão, o **Matlaber** cumpriu os requisitos, partindo de um programa em IL e permitindo simulá-lo em ambiente *Simulink*, onde se encontrava modelado o processo em estudo.

6.2.2 SIMULAÇÃO 2: SERRA

Para 2ª simulação, foi utilizado um modelo de um sistema linear pré-existente. Consiste numa serra circular de bancada (Figura 6.11) que executa um movimento cíclico controlado por CLP. Possui vários indicadores de posição, que estão representados por um interruptor de contacto e são denominados S1 a S5, com o botão de acionamento *Start* a completar o modelo.

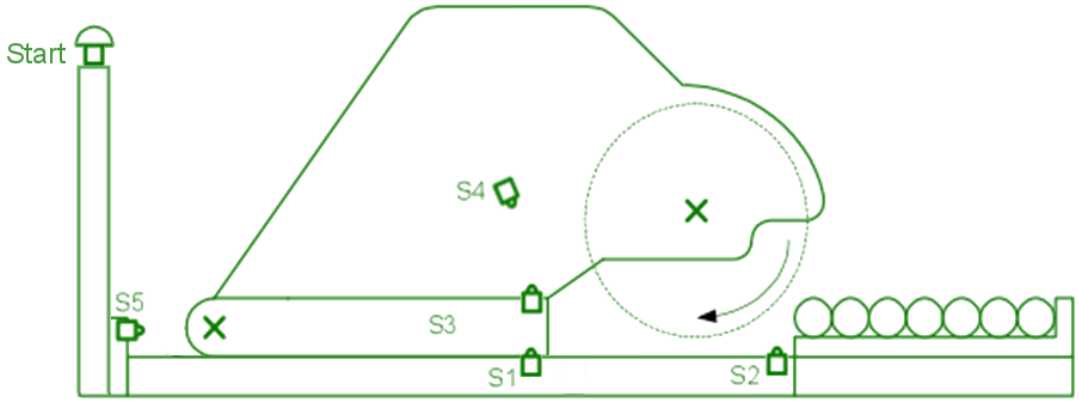

Figura 6.11 - Representação do modelo da serra

A serra circular deverá ser ativada (posta em rotação) unicamente quando na zona de corte, erguida seguidamente e reposta na sua posição inicial. O andamento cíclico do processo de corte está representado na Figura 6.12. Na mesma figura, está representado o SFC de controlo, lado a lado com as posições e ações associadas. O seu controlo sequencial está ainda acompanhado da representação sequencial do andamento físico da máquina. Os valores de entradas e saídas estão indicados pelos endereços I/O sendo detalhados na Tabela 6.2.

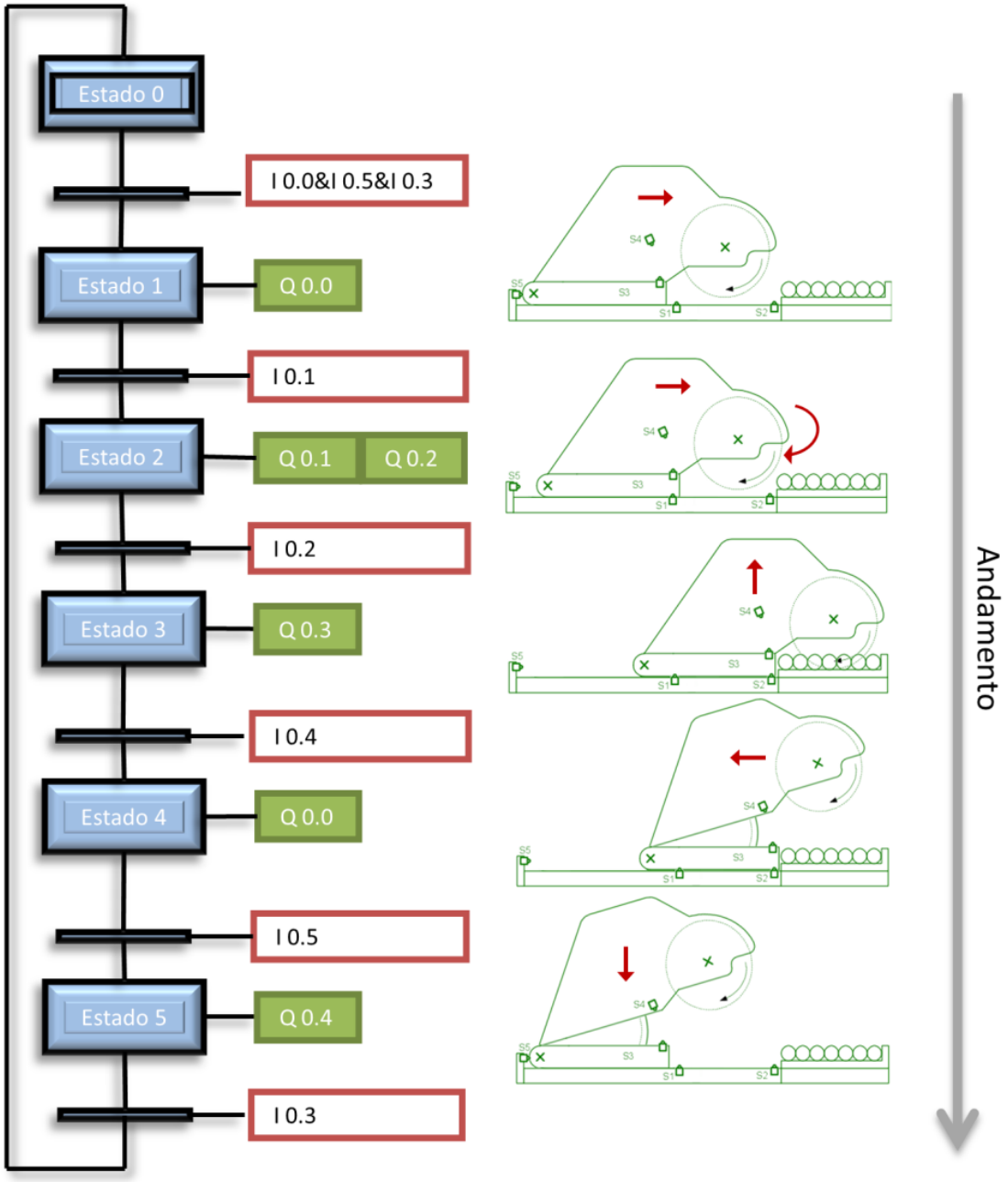

Figura 6.12 - Diagrama funcional (SFC) da simulação 2 "Serra"

O programa de controlo foi modelado segundo o SFC da Figura 6.12 e resultou no código de CLP proprietário, em *Siemens* STL, cuja listagem completa pode ser consultada na Figura 6.13.

```
Network 1//SFC start
LD         M0.5//start
program
A       I0.3
LD    M0.0
AN   M0.1
OLD
=       M0.0
Network 2
LD    M0.0
A       I0.0
A       I0.5
A       I0.3
LD    M0.1
AN    M0.2
OLD
=       M0.1
Network 3
LD    M0.1
A       I0.1
LD    M0.2
AN    M0.3
OLD
=     M0.2
Network 4
LD    M0.2
A       I0.2
LD    M0.3
AN    M0.4
OLD
=       M0.3
```

```
Network 5
LD    M0.3
A       I0.4
LD    M0.4
AN    M0.5
OLD
=       M0.4

Network 6
LD    M0.4
A       I0.5
LD    M0.5
AN    M0.0
OLD
=  M0.5 // end SFC

Network  7//Write  the
outputs
LD    M0.1
LD    M0.2
OLD
=    Q0.1  //go right

Network 8
LD    M0.2
=    Q0.2//spin saw

Network 9
LD    M0.3
=    Q0.3
Network  10    //raise
machine

LD    M0.4
=    Q0.0 //go left

Network 11
LD    M0.5
=     Q0.4     //lower
machine

//End of program
```

Figura 6.13 - Código STL do programa de controlo da simulação 2

O programa de controlo apenas faz recurso a instruções de lógica booleana que, através das variáveis de estado M, que representam os vários estados do SFC e lendo os valores dos sensores, fazem o programa avançar no seu estado quando pertinente e gerando as saídas de controlo da serra.

Criando a montagem, em ambiente *Simulink*, do modelo em estudo previamente modelado, obteve-se a representação da Figura 6.14. O bloco "*Sawmill process*" encontra-se já modelado e representa a serra circular, interruptores de posição e motores de atuação. A representação segue

o modelo generalista indicado no capítulo 5.1.3. Alterações de nota incluem os blocos: botão de início de processo ("*START Pushbuton1*"), o gerador de relógio ("*Clock*"), e vários mostradores.

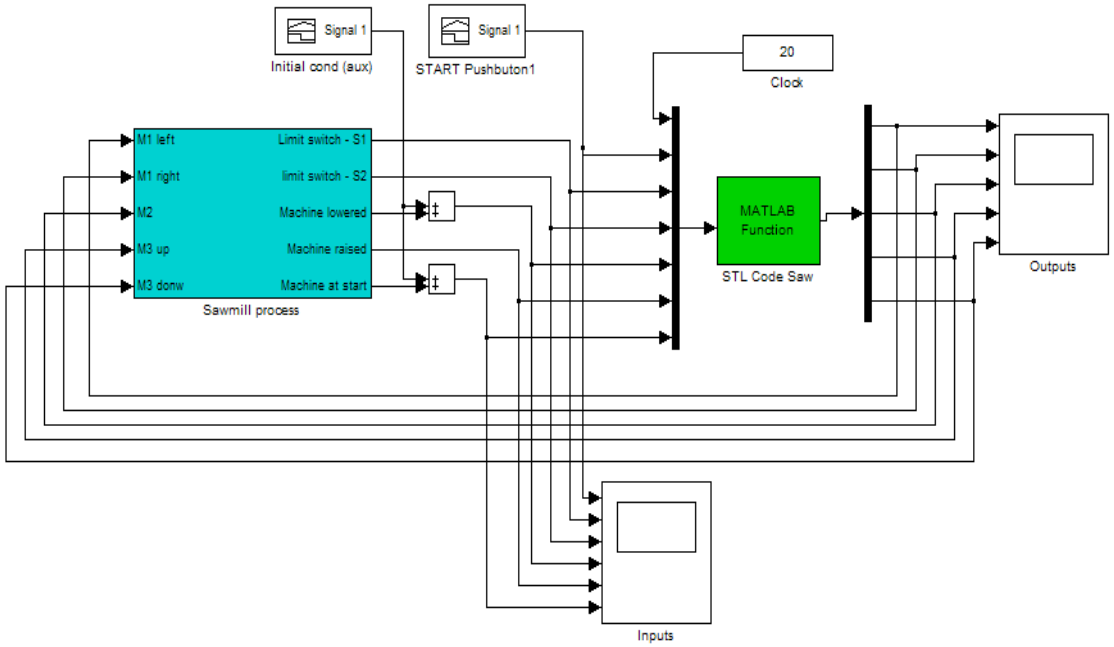

Figura 6.14 – Representação em *Simulink* de simulação da serra

É então necessário modelar o programa de controlo da Figura 6.13 para a forma proposta pelo presente trabalho. Dado que o programa de controlo se encontra escrito numa linguagem proprietária, será utilizada a combinação da tradução do programa de controlo para IL normalizado, mais a conversão para *Matlab*, recorrendo ao **UnifIL** e ao **Matlaber**, respetivamente.

Começando pela tradução, o código do programa de controlo original foi traduzido, através do sistema computacional desenvolvido, o **UnifIL**, para código IL normalizado (Figura 6.15), utilizando para isso o dicionário de regras de tradução para *Siemens* STL desenvolvido.

```
(* START PROGRAM *)
LD      M0_5
AND     I0_3
OR(     M0_0
ANDN M0_1
)
ST      M0_0
LD      M0_0
AND     I0_0
AND     I0_5
AND     I0_3
OR(     M0_1
ANDN M0_2
)
ST      M0_1
LD      M0_1
AND     I0_1
OR(     M0_2
ANDN M0_3
)
ST      M0_2
LD      M0_2
AND     I0_2
OR(     M0_3
ANDN M0_4
)
ST M0_3
LD M0_3
AND I0_4
OR( M0_4
ANDN M0_5
)
ST M0_4
LD M0_4
AND I0_5
OR( M0_5
ANDN M0_0
)
ST M0_5  (*  END SFC *)

(* WRITE THE OUTPUTS *)

LD M0_1
OR( M0_2
)
ST Q0_1   (* GO RIGHT *)

LD M0_2
ST Q0_2 (* SPIN SAW *)

LD M0_3
ST Q0_3
LD M0_4
ST Q0_0  (* GO LEFT *)

LD M0_5
ST Q0_4
(* LOWER MACHINE *)

(* END OF PROGRAM *)

END_PROGRAM
```

Figura 6.15 - Código funcional em IL normalizado do programa de controlo da simulação 2

O código ficou mais compacto e legível, algumas das virtudes inerentes ao IL do IEC 61131-3. Este código não inclui cabeçalho nem declaração de variáveis, para fins de clareza. A listagem completa pode ser consultada no Anexo XVII.

Seguidamente, é necessário converter o código para modelo *Matlab*, com recurso ao **Matlaber.** O código IL da Figura 6.15 foi convertido para ficheiro ".m", ficando então pronto para integração com o modelo do processo em ambiente *Matlab*/*Simulink*. O código resultante pode ser observado na Figura 6.16. A listagem completa pode ser consultada no Anexo XVIII.

```
M0_0 = ((M0_5&I0_3)|(M0_0&~M0_1)); %  START PROGRAM ||
M0_1 = ((((M0_0&I0_0)&I0_5)&I0_3)|(M0_1&~M0_2));
M0_2 = ((M0_1&I0_1)|(M0_2&~M0_3));
M0_3 = ((M0_2&I0_2)|(M0_3&~M0_4));
M0_4 = ((M0_3&I0_4)|(M0_4&~M0_5));
M0_5 = ((M0_4&I0_5)|(M0_5&~M0_0)); %   END SFC  ||

%  WRITE THE OUTPUTS  ||
Q0_1 = M0_1|(M0_2); %  GO RIGHT  ||
Q0_2 = M0_2; %  SPIN SAW  ||
Q0_3 = M0_3;
Q0_0 = M0_4; %  GO LEFT  ||
Q0_4 = M0_5; %  LOWER MACHINE  ||

% END OF PROGRAM  ||

%Output generation
OUTPUT_VECTOR=[Q0_0*1,Q0_1*1,Q0_2*1,Q0_3*1,Q0_4*1];
```

Figura 6.16 - Código funcional de controlo convertido para Matlab

Após uma breve análise a este código, a lógica subjacente a cada estado e saída torna-se muito mais aparente no código convertido, fruto da conversão simbólica descrita no capítulo 5.1.4. As correspondências das entradas e saídas (todas lógicas) podem ser consultadas na Tabela 6.2.

Tabela 6.2 - Descrição dos sinais, ações e endereços

Descrição do sinal	Descrição da ação	Endereço *I/O* no CLP	Variável correspondente em *Matlab/Simulink*
Botão de interface de controlo	Iniciar	I0.0	I0_0
Interruptor de posição – S1	Serra na posição média	I0.1	I0_2
Interruptor de posição – S2	Serra na posição avançada	I0.2	I0_3
Interruptor de posição – S3	Cabeça em posição baixa	I0.3	I0_4
Interruptor de posição – S4	Cabeça em posição alta	I0.4	I0_5
Interruptor de posição – S5	Serra na posição inicial	I0.5	I0_6
Motor M1 - Esquerda	Mover para a esquerda	Q0.0	Q0_1
Motor M1 - Direita	Mover para a direita	Q0.1	Q0_2
Motor M2	Rodar serra	Q0.2	Q0_3
Motor M3 - Cima	Subir cabeça	Q0.3	Q0_4
Motor M3 - Baixo	Descer cabeça	Q0.4	Q0_5

É de notar a transformação dos endereços das variáveis para a sua versão em *Matlab* como descrito no capítulo 5.3.3.2. Correndo a simulação, obtiveram-se os resultados dos sensores e atuadores ilustrados na Figura 6.17.

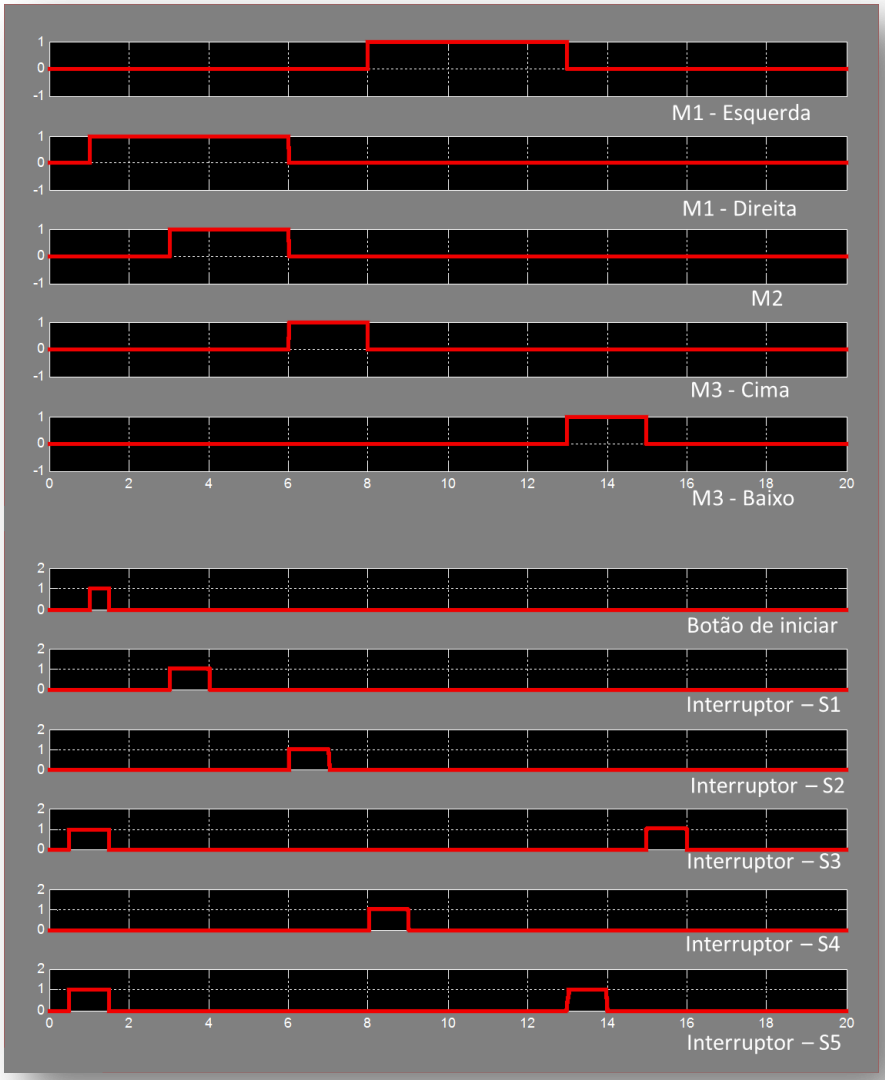

Figura 6.17 - Resultados da simulação da simulação 2

Devido à natureza do processo e análise exclusiva através dos valores dos sensores e atuadores, não é imediatamente aparente o funcionamento correto da simulação. Mas, analisando estes valores e comparando-os com os valores definidos pelo SFC da Figura 6.12, é possível discernir que o funcionamento é exatamente o esperado.

Tendo isso em conta, o programa de controlo funciona perfeitamente, cumprindo todos os requisitos a que se propõe. Por esta análise de tradução e conversão, tanto o **UnifIL** como o **Matlaber** cumpriram os requisitos, partindo de um programa de CLP em STL e permitindo simulá-lo em ambiente *Simulink*, onde se encontrava o modelo do processo em estudo.

7 CONCLUSÕES E TRABALHO FUTURO

Apresenta-se uma reflexão sobre o trabalho desenvolvido, uma análise do cumprimento dos objetivos apontados, assim como uma antevisão de trabalhos futuros e expansões possíveis.

7.1 REFLEXÃO

Na atualidade são utilizados CLP para controlar os mais variados processos fabris. Mas um dos grandes problemas são os testes associados ao desenvolvimento de um programa de controlo. O trabalho desenvolvido pretende possibilitar a análise, teste e validação do programa de controlo que será imposto no CLP.

Para o cumprimento desse objetivo, propôs-se um método que permitisse utilizar os programas de controlo originais para controlar um modelo simulado do processo industrial em estudo. A ideia central do trabalho apresentado é a de modelar o próprio programa de controlo para uma forma que o ambiente de simulação alvo reconhecesse nativamente.

Mais pragmaticamente, o trabalho desenvolvido oferece um conjunto complementar de 2 sistemas computacionais que permitem simular, em ambiente *Matlab*/*Simulink*, o controlo por CLP de um processo industrial modelado. O controlo pode ser, preferencialmente, programado na abrangente linguagem IL da norma vigente IEC 61131. Apesar da crescente abrangência do suporte às linguagens da norma e da larga abrangência de uso do IL, o suporte a outras linguagens continua a ser crucial. Portanto, o controlo pode ser programado em várias linguagens proprietárias, expansíveis.

Os 2 sistemas e respetivas funcionalidades oferecidas são:

- **Matlaber –** converte os códigos na linguagem IL, da norma IEC 61131, para ficheiros ".m" reconhecidos no ambiente *Matlab*/*Simulink*.
- **UnifIL –** normaliza códigos em várias linguagens CLP proprietárias para IL normalizado. O sistema oferece suporte, de raiz, a 2 linguagens proprietárias (*Siemens S200* STL e *Mitsubishi Q Series*), mas é ainda expansível a outras linguagens através da criação de dicionários de regras de tradução especializados, que são escritos em XML , de uma forma aberta e acessível.

Ambos os sistemas computacionais cumprem os objetivos a que se propuseram, completando o método proposto pelo trabalho apresentado. O método desenvolvido apresenta ainda várias vantagens, nem todas antevistas aquando da sua formulação, nomeadamente:

- O uso da linguagem da norma IEC 61131-3 acabou por se revelar frutífero, pois com muitas instruções simples (afinal, esta linguagem é próxima do *assembly*) conseguem-se codificar programas de controlo complexos sem necessidade de sair do ambiente de simulação para poder replicar esse funcionamento complexo. Os exemplos de simulação são espelho desta característica.
- A expansibilidade dos dicionários de tradução é mais simples do que o esperado para linguagens que tenham poucas diferenças.
- A abrangência de códigos em IL é grande mas, mesmo sem a implementação completa de todas as instruções definidas pelo IEC 61131-3, o bom suporte às instruções fundamentais revelou-se completamente satisfatório, pois 69% das instruções utilizadas em CLP são de lógica booleana [53], o que não inclui instruções de fluxo de programa: ambas as categorias são suportadas a 100% pelo trabalho desenvolvido.
- O método desenvolvido permite usar o IDE do fabricante. Os avanços dos novos IDEs, ou qualquer outro meio de programação de CLP, continuarão a beneficiar programação de controlo por CLP que recorra ao trabalho apresentado para testes, simulações e validações.
- O uso recursivo de simulações com programas de controlo diferentes revelou-se fluído e intuitivo. Como o bloco de *Simulink*, que representa o CLP na simulação, chama um ficheiro externo (o ficheiro ".m"), este fica imediatamente pronto a ser utilizado, sem nenhuma intervenção no *Simulink* necessária, aquando da atualização da conversão.

O trabalho desenvolvido foi ainda apresentado e publicado na conferência ISIE 2011, através do artigo científico: "*The use of IEC 61131-3 to enhance CLP control and Matlab/Simulink process simulations*" [54].

7.2 TRABALHO FUTURO

O trabalho apresentado deixou algumas características que não foram totalmente implementadas e conceitos inexplorados, que poderão ser atacados de futuro, nomeadamente:

- Dicionários de regras de tradução – É possível completar os dicionários de tradução existentes para um suporte mais completo das linguagens suportadas de origem, assim

como a criação de dicionários completamente novos para expandir ainda mais a abrangência de linguagens proprietárias.

- Suporte às instruções em falta do IL – Instruções numéricas e aritméticas foram apenas parcialmente implementadas, enquanto que instruções de conversões de tipos, operações de *strings* e instruções de *bit-shift* não foram implementadas.
- Suporte total de funções e dos POU – Para um suporte completo, é possível complementar a implementação inicial com os POU de *timers* e funções ausentes.
- Suporte a comunicação – De acordo com a norma IEC 61131-5.

8 BIBLIOGRAFIA

[1] D. F. Noble, *Forces of Production: A Social History of Industrial Automation*, U.S.A.: Knopf, 1984.

[2] M. P. Groover, *Automation, Production Systems, and Computer Integrated Manufacturing*, Prentice-Hall, 1987.

[3] A. Kusiak, *Computational Intelligence in Design and Manufacturing*, John Wiley AND Sons, 2000.

[4] S. B. Morriss, *Automated Manufacturing Systems*, McGraw Hill, 1994.

[5] Modelo Bearbeitungseinheit, http://www.staudinger-est.de/, acedido em 2011-04-15.

[6] V. Pinto, S. Rafael, J. F. Martins, "PLC controlled industrial processes on-line simulator", IEEE International Symposium on Industrial Electronics, ISIE 2007, junho 2007, Vigo, España.

[7] PC-SIM, http://www.autoware.com/english/pc-sim.htm/, acedido em 2010-10-29.

[8] PSIM, http://www.thelearningpit.com/, acedido em 2010-11-12.

[9] D. Friedrich, B. Vogel-Heuser, "Benefit of system modeling in automation and control education", American Control Conference, New York, U.S.A., 2007.

[10] T. Deveza, J. F. Martins, "PLC Control and Matlab/Simulink Simulations. A Translation Approach", IEEE International Conference on Emerging Technology and Factory Automation, ETFA 2009, setembro 2099, Palma de Mallorca, España.

[11] Mathworks Matlab/Simulink, http://www.mathworks.com, acedido em 2010-11-20.

[12] "IEC 61131-3 Programmable controllers Part 3: Programming Languages", IEC, 2003.

[13] J. Karl-Heinz, M. Tiegelkamp, IEC *61131-3: Programming Industrial Automation Systems 2nd Edition*, Springer, 2010.

[14] G. L. Kim, P. Paul, Y. Wang, "UPPAAL in a nutshell", International Journal on Software Tools for Technology Transfer,1997.

[15] International Electrotechnical Commission, http://www.iec.ch/, acedido em 2010-11-20.

[16] International Electrotechnical Commission, IEC *61131-3 2nd Edition*, IEC publications, 2003.

[17] G. Turnbull, *Open-ness and* IEC *61131-3*, The Institution of Electrical Engineers, 2002.

[18] A. J. Wilson, T. Hill, *An industrial case study covering the use of* IEC *1131-3 in connection with the engineering and operation requirements of the water industry*, The Institution of Electrical Engineers, 1999.

[19] G. Turnbull, *The importance of 61131 to UK Business*, The Institution of Electrical Engineers, 1999.

[20] M. de Sousa, *Restricting* IEC *61131-3 Programming Languages for use on High Integrity Applications*, Emerging Technologies and Factory Automation IEEE International Conference, Hamburg, Deutschland, 2008.

[21] M. Wenger, A. Zoitl, C. Sünder, H. Steininger, *Semantic correct transformation of* IEC *61131-3 models into the IEC 61499 standard*, Emerging Technologies and Factory Automation IEEE International Conference, Palma de Mallorca, España, 2009.

[22] I. Plaza, C. Medrano, *A specific implementation of* IEC *61131-3 software model*, IEEE World Automation Congress (WAC 2004), Seville, España, 2004.

[23] N. Bauer, R. Huuck, B. Lukoschus, S. Engell, *A Unifying Semantics for Sequential Function Charts*, Integration of Software Specification Techniques for Applications in Engineering, LNCS, 2004.

[24] Alex Morris, George Oluwande, IEC*61131 – A users' perspective from Innogy*, IEE Colloquium on The Application of IEC 61131 in Industrial Control: Improve Your Bottom-line Through High Value Industrial Control Systems, London, U.K., 2002.

[25] E. Estévez, M. Marcos, E.Irisarri, F. Lopéz, I. Sarachaga, A. Burgos, *A novel Approach to attain the true reusab*IL*ity of the code between diferent* PLC *programming Tools*, IEEE International Workshop on Factory Communication Systems, Dresden, Deutschland, 2008.

[26] F.J. Molina, J. Barbancho, C. Leon, A. Molina, A. Gomez, *Using Industrial Standards on* PLC *Programming Learning*, Mediterranean Conference on Control and Automation, Καλαμάτα (Kalamata), Ελλάδα (Grécia), 2007.

[27] Andy Verwer, *The impact of* IEC *(6)1131-3 on the teaching of control engineering*, IEE Colloquium on the Application of IEC 61131 to Industrial Control: Improve Your Bottom Line Through High Value Industrial Control Systems, 1999.

[28] PLCOpen Editor, www.PLCopen.org/, acedido em 2010-11-21.

[29] eXtensible Markup Language (XML) 1.1 (Second Edition), http://www.w3.org/TR/2006/REC-xml11-20060816/

[30] http://www.automation.siemens.com/, acedido em 2010-11-23.

[31] www.schneider-electric.com/ acedido em 2010-11-23.

[32] http://www.kw-software.com/, acedido em 2010-11-23.

[33] http://www.3s-software.com/, acedido em 2010-11-23.

[34]http://www.boschrexroth.com/, acedido em 2010-11-23.

[35] http://www.sta-gmbh.ch/, acedido em 2010-11-23.

[36] http://www.moeller.co.uk/xsoft.htm/, acedido em 2010-10-29.

[37] http://www.wago.us/, acedido em 2010-10-29.

[38] http://www.beremiz.org/, acedido em 2011-05-19.

[39] OpenPCS Automation Suite, http://www.infoteam.de/, acedido em 2011-05-19.

[40] Simulink PLC Coder, http://www.mathworks.com/products/sl-plc-coder/, acedido em 2011-06-16.

[41] Unified Modeling Language specification, http://www.uml.org/, acedido em 2011-05-19.

[42] Y. Yan, H. Zhang, *Compiling Ladder Diagram into Instruction List to comply with* IEC *61131-3*, Computers in Industry, Volume 61, Issue 5, June 2010.

[43] G. Fen, W. Ning, A Transformation Algorithm of Ladder Diagram into Instruction List Based on AOV Digraph and Binary Tree, IEEE Tencon Region 10 Conference, 香港 (Hong Kong), 中华人民共和国(China), 2006.

[44] J. Pires, S. Ge, T. Lee, D. Gu, L. Woon, *Interfacing industrial R&A equipment using Matlab*, Robotics & Automation Magazine, IEEE, Volume 7, Issue 3, 2000.

[45] Modelica, https://www.modelica.org/, acedido em 2011-05-19.

[46] L. Nagel, D. Pederson, *SPICE (Simulation Program with Integrated Circuit Emphasis)*, Memorandum, University of California, Berkeley, 1973.

[47] ASPEN Plus, http://www.aspentech.com/, acedido em 2011-05-19.

[48] E. Carpanzano, A. Ballarino, *A Structured Approach to the Design and Simulation-based Testing of Factory Automation Systems*, Institute of lndustrial Technologies & Automation, National Research Council, Itália, 2002.

[49] Microsoft Visual Studio, http://www.microsoft.com/visualstudio/en-us/, acedido em 2011-05-19.

[50] Visual Paradigm for UML, http://www.visual-paradigm.com/, acedido em 2011-06-27.

[51] F. Rubin, "GOTO considered harmful" Considered Harmful, *Communications of the ACM*, 1987.

[52] H. Aldahiyat, "MATLAB Goto Statement", Matlab Central tools' library, livre para fins não lucrativos, http://www.mathworks.com/matlabcentral/fileexchange/26949, acedido em 2010-11-25.

[53] G. Rho, K. Koo, N. Chang, J. Park, Y. Kim, W. Kwon, *Implementation of a RISC microprocessor for programmable logic controllers*, Elsevier Science B.V, Microprocessors and Microsystems, Volume 19, Number 10, dezembro 1995.

[54] A. Pereira, C. Lima, J. F. Martins, "The use of IEC 61131-3 to enhance PLC control and Matlab/Simulink process simulations", 20th International Symposium on Industrial Electronics 2011, Gdańsk, Polska (Polónia), janeiro 2011.

Anexo I

Tabela de funções suportadas pelo IEC 61131-3

Standard functions (with data types of input variables)	Data type of function value	Short description	over-loa-ded	exten-sible
Type conversion				
_TO_* (ANY)	ANY	Data type conversion	yes	no
TRUNC (ANY_REAL)	ANY_INT	Rounding up/down	yes	no
BCD_TO_** (ANY_BIT)	ANY	Conversion from BCD	yes	no
***_TO_BCD** (ANY_INT)	ANY_BIT	Conversion to BCD	yes	no
DATE_AND_TIME_TO_-TIME_OF_DAY (DT)	TOD	Conversion to time-of-day	no	no
DATE_AND_TIME_TO_-DATE (DT)	DATE	Conversion to date	no	no
Numerical				
ABS (ANY_NUM)	ANY_NUM	Absolute number	yes	no
SQRT (ANY_REAL)	ANY_REAL	Square root (base 2)	yes	no
LN (ANY_REAL)	ANY_REAL	Natural logarithm	yes	no
LOG (ANY_REAL)	ANY_REAL	Logarithm to base 10	yes	no
EXP (ANY_REAL)	ANY_REAL	Exponentiation	yes	no
SIN (ANY_REAL)	ANY_REAL	Sine	yes	no
COS (ANY_REAL)	ANY_REAL	Cosine	yes	no
TAN (ANY_REAL)	ANY_REAL	Tangent	yes	no
ASIN (ANY_REAL)	ANY_REAL	Arc sine	yes	no
ACOS (ANY_REAL)	ANY_REAL	Arc cosine	yes	no
ATAN (ANY_REAL)	ANY_REAL	Arc tangent	yes	no
Arithmetic ***(IN1, IN2)***				
ADD {+} (ANY_NUM, ANY_NUM)	ANY_NUM	Addition	yes	yes
ADD {+} [a] (TIME, TIME)	TIME	Time addition	yes	no
ADD {+} [a] (TOD, TIME)	TOD	Time-of-day addition	yes	no
ADD {+} [a] (DT, TIME)	DT	Date addition	yes	no
MUL {*} (ANY_NUM, ANY_NUM)	ANY_NUM	Multiplication	yes	yes
MUL {*} [a] (TIME, ANY_NUM)	TIME	Time multiplication	yes	no
SUB {-} (ANY_NUM, ANY_NUM)	ANY_NUM	Subtraction	yes	no
SUB {-} [a] (TIME, TIME)	TIME	Time subtraction	yes	no
SUB {-} [a] (DATE, DATE)	TIME	Date subtraction	yes	no
SUB {-} [a] (TOD, TIME)	TOD	Time-of-day subtraction	yes	no
SUB {-} [a] (TOD, TOD)	TIME	Time-of-day subtraction	yes	no
SUB {-} [a] (DT, TIME)	DT	Date and time subtraction	yes	no
SUB {-} [a] (DT, DT)	TIME	Date and time subtraction	yes	no
DIV {/} (ANY_NUM, ANY_NUM)	ANY_NUM	Division	yes	no
DIV {/} [a] (TIME, ANY_NUM)	TIME	Time division	yes	no
MOD (ANY_NUM, ANY_NUM)	ANY_NUM	Remainder (modulo)	yes	no
EXPT {**} (ANY_NUM, ANY_NUM)	ANY_NUM	Exponent	yes	no
MOVE {:=} (ANY_NUM, ANY_NUM)	ANY_NUM	Assignment	yes	no

a Special function for time data type. The generic input and output data type for ADD and SUB is therefore ANY_MAGNITUDE, see also Section 3.4.3

Tabela de funções suportadas pelo IEC 61131-3 (continuação)

Standard functions (with data types of input variables)		Data type of function value	Short description	over-loa-ded	exten-sible
Bit-shift	***(IN1, N)***				
SHL	(ANY_BIT, ANY_INT)	ANY_BIT	Shift left	yes	no
SHR	(ANY_BIT, ANY_INT)	ANY_BIT	Shift right	yes	no
ROR	(ANY_BIT, ANY_INT)	ANY_BIT	Rotate right	yes	no
ROL	(ANY_BIT, ANY_INT)	ANY_BIT	Rotate left	yes	no
Bitwise	***(IN1, IN2)***				
AND {&}	(ANY_BIT, ANY_BIT)	ANY_BIT	Bitwise AND	yes	yes
OR {>=1}	(ANY_BIT, ANY_BIT)	ANY_BIT	Bitwise OR	yes	yes
XOR {=2k+1}	(ANY_BIT, ANY_BIT)	ANY_BIT	Bitwise EXOR	yes	yes
NOT	(ANY_BIT, ANY_BIT)	ANY_BIT	Bitwise inverting	yes	no
Selection	***(IN1, IN2)***				
SEL	(G, ANY, ANY)	ANY	Binary selection (1 of 2)	yes	no
SEL [b]	(G, ENUM, ENUM)	ENUM	Binary selection (1 of 2)	no	no
MAX [c]	(ANY_E, ANY_E)	ANY_E	Maximum	yes	yes
MIN [c]	(ANY_E, ANY_E)	ANY_E	Minimum	yes	yes
LIMIT[c]	(MN, ANY_E, MX)	ANY_E	Limitation	yes	no
MUX[c]	(K, ANY, ..., ANY)	ANY	Multiplexer (select 1 of N)	yes	yes
MUX [b]	(K, ENUM, ..., ENUM)	ENUM	Multiplexer (select 1 of N)	no	no
Comparison	***(IN1, IN2)***				
GT {>}	(ANY, ANY)	BOOL	Greater than	yes	yes
GE {>=}	(ANY, ANY)	BOOL	Greater than or equal to	yes	yes
EQ {=}	(ANY, ANY)	BOOL	Equal to	yes	yes
EQ {=} [b]	(ENUM, ENUM)	BOOL	Equal to	no	no
LT {<}	(ANY, ANY)	BOOL	Less than	yes	yes
LE {<=}	(ANY, ANY)	BOOL	Less than or equal to	yes	yes
NE {<>}	(ANY, ANY)	BOOL	Not equal to	yes	no
NE {<>} [b]	(ENUM, ENUM)	BOOL	Not equal to	no	no
Character string	***(IN1, IN2)***				
LEN	(STRING)	INT	Length of string	no	no
LEFT	(STRING, L)	STRING	string “left of”	yes	no
RIGHT	(STRING, L)	STRING	string “right of”	yes	no
MID	(STRING, L, P)	STRING	string “from the middle”	yes	no
CONCAT	(STRING, STRING)	STRING	Concatenation	no	yes
CONCAT [a]	(DATE, TOD)	DT	Time concatenation	no	no
INSERT	(STRING, STRING, P)	STRING	Insertion (into)	yes	yes
DELETE	(STRING, L, P)	STRING	Deletion (within)	yes	yes
REPLACE	(STRING, STRING, L, P)	STRING	Replacement (within)	yes	yes
FIND	(STRING, STRING)	INT	Find position	yes	yes

a Special function for time data type
b Special function for enumeration data type
c ANY_E is the abbreviation of ANY_ELEMENTARY

Anexo II

Tabela de blocos funcionais definidos pelo IEC 61131-3

Name of std. FB with input parameter names	Names of output parameters	Short description
Bistable elements		
SR (S1, R,	Q1)	Set dominant
RS (S, R1,	Q1)	Reset dominant
Edge detection		
R_TRIG {->} (CLK,	Q)	Rising edge detection
F_TRIG {-<} (CLK,	Q)	Falling edge detection
Counters		
CTU (CU, R, PV,	Q, CV)	Up counter
CTD (CD, LD, PV,	Q, CV)	Down counter
CTUD (CU, CD, R, LD, PV,	QU, QD, CV)	Up/down counter
Timers		
TP (IN, PT,	Q, ET)	Pulse
TON {T---0} (IN, PT,	Q, ET)	On-delay
TOF {0---T} (IN, PT,	Q, ET)	Off-delay
Communication		***See IEC 61131-5***

Tabela de significados dos parâmetros de entrada e saída

Inputs / Outputs	Meaning	Data type
R	Reset input	BOOL
S	Set input	BOOL
R1	Reset dominant	BOOL
S1	Set dominant	BOOL
Q	Output (standard)	BOOL
Q1	Output (flip-flops only)	BOOL
CLK	Clock	BOOL
CU	Input for counting up	R_EDGE
CD	Input for counting down	R_EDGE
LD	Load (counter) value	INT
PV	Pre-set (counter) value	INT
QD	Output (down counter)	BOOL
QU	Output (up counter)	BOOL
CV	Current (counter) value	INT
IN	Input (timer)	BOOL
PT	Pre-set time value	TIME
ET	End time output	TIME
PDT	Pre-set date and time value	DT
CDT	Current date and time	DT

Anexo III

Tabela dos tipos de variáveis suportadas pelo conversor

Variáveis binárias	BOOL, BYTE, WORD, LWORD, DWORD
Variáveis inteiras	INT, DINT, LINT, USINT, UINT, UDINT, ULINT, REAL, LREAL
Variáveis temporais	DATE, TOD, DT, TIME
Variáveis de texto	STRING, WSTRING

Anexo IV

Listagem do código fonte do método *ReadProgramTitle*

```
class_types.read_function_output ReadProgramTitle(string code, class_types.m_file_data m_file_input, int current_line)
        {
            class_types.read_function_output result;
            class_types.CodeResult TempCodeComment;
            int wrong_spaces = 0;
            int name_length = 0;
            result.lines_forward = 1;
            result.success = false;
            result.message = "";
            result.m_file = m_file_input;
            //comments "(* comment *)" removal and adding
            TempCodeComment = RemoveComment(code);

            if (TempCodeComment.success == false)//check for malformed comments
            {
                result.success = false;
                result.message += "Error in comments at line " + (current_line + 1) + ". ";
                return result;
            }

            result.m_file.title.comment = TempCodeComment.comment;
            code = TempCodeComment.code;

            while (code.Substring(wrong_spaces, 1) == " " && wrong_spaces < code.Length)//clean up wrong spaces
                wrong_spaces++;

            if (wrong_spaces > 0)
                result.message += "Warning: There extra spaces in line " + (current_line + 1) + ". Matlaber was able to
continue, but you should fix this. ";
            if (code.Length < 9)
            {
                result.success = false;
                result.message = "Error: Invalid code. ";
            }
            else
            {
                if (code.Substring(wrong_spaces, 8) == "PROGRAM ")
                {
                    if (code.Substring(9 + wrong_spaces, code.Length - 9 - wrong_spaces).Contains(" "))
                        name_length = code.IndexOf(" ", 9) - 8 - wrong_spaces;
                    else
                        name_length = code.Length - 8 - wrong_spaces;

                    result.m_file.title.code = code.Substring(8 + wrong_spaces, name_length);
                    result.success = true;

                }
                else //in case PROGRAM instruction is not found
                {
                    result.success = false;
                    result.message = "Error: PROGRAM instruction not found. ";
                }
            }
            return result;
        }
```

Anexo V

Listagem do código fonte do método *ReadVariables*

```
class_types.read_function_output ReadVariables(LinkedList<string> InCode, class_types.m_file_data m_file_input, int
current_line)// function to read a block of variables
        {
            class_types.read_function_output result;
            class_types.CodeResult TempCodeComment;
            class_types.m_variable var_temp = new class_types.m_variable();
            int wrong_spaces = 0;
            string IO_type = "";
            int colon_index, semicolon_index, colon_2_index = 0;
            result.lines_forward = 1;
            result.success = false;
            result.message = "";
            result.m_file = m_file_input;
            TempCodeComment = RemoveComment(InCode.First());
            if (TempCodeComment.success == false)//check for malformed comments
            {
                result.success = false;
                result.message += "Error in comments at line " + (current_line + current_line ) + ". ";
                return result;
            }//var block's comment line is here
            string temp =TempCodeComment.comment;
            InCode.RemoveFirst();
            InCode.AddFirst(TempCodeComment.code);
            while (InCode.First() == "")//account for empty lines
            {
                InCode.RemoveFirst();
                result.lines_forward++;
                if (InCode.Count <= 0) //in case the EOF is encountered while searching for non-empty lines
                {   result.success = false;
                    result.message += "Sudden end of file at line " + (current_line + result.lines_forward + 1) + ". ";
                    return result;}
            })//clean up wrong spaces
            while (InCode.First().Substring(wrong_spaces, 1) == " " && wrong_spaces < InCode.First().Length
               wrong_spaces++;
            int IO_case = -1; //code line clean, onto var block definition (IO type)
            int before_retention = 0;
            class_types.m_variable var_temp_vector = new class_types.m_variable();
                if (InCode.First().Length > 8 && InCode.First().Substring(0, 9) == "VAR_INPUT")
                    IO_case = 0;
                else
                {   if (InCode.First().Length > 9 && InCode.First().Substring(0, 10) == "VAR_OUTPUT")
                        IO_case = 1;
                    else
                    {
                        if (InCode.First().Length > 9 && InCode.First().Substring(0, 10) == "VAR_GLOBAL")
                            IO_case = 2;
                        else
                        {
                            if (InCode.First().Length > 2 && InCode.First().Substring(0, 3) == "VAR")
                             IO_case = 2;
                        }
                    }}
                switch (IO_case)
                {   case 0:
                        IO_type = "input";
                        before_retention = 8;
                        break;
                    case 1:
                        IO_type = "output";
                        before_retention = 9;
                        break;
                    case 2:
                        IO_type = "global";
                        before_retention = 9;
                        break;}
            if (InCode.First().Length > (before_retention + 7))//Check for retention parameters
            {
                wrong_spaces = 0;
                while (InCode.First().Substring(wrong_spaces + before_retention + 1, 1) == " " && wrong_spaces <
InCode.First().Length - before_retention - 1)//clean up wrong spaces
                    wrong_spaces++;
                if (InCode.First().Substring(wrong_spaces + before_retention + 1, 6) == "RETAIN")
                    var_temp.retain = true;
                else
                    var_temp.retain = false;
            }
            if (IO_type == "") //case no type of variable is understood
            {   result.success = false;
                result.message = result.message + "Error: invalid variable IO option at line  " + (current_line +
result.lines_forward + 1) + ". ";
                return result;}
            else
            {if (temp!="") //complete var block's comment
                    result.m_file.var_block_comments.AddLast(IO_type + " block's comment: " + temp);}
            InCode.RemoveFirst();
            result.lines_forward++;
            //account for empty lines
            while (InCode.First() == "")
            {
```

```
                InCode.RemoveFirst();
                result.lines_forward++;
                if (InCode.Count <= 0) //in case the EOF is encountered while searching for non-empty lines
                {
                    result.success = false;
                    result.message += "Sudden end of file at line " + (current_line + result.lines_forward + 1) + ". ";
                    return result;
                }
            }//vars declaration loop
            while (InCode.First() != "" && InCode.First().Substring(0, 7) != "END_VAR")
            {
                wrong_spaces = 0;
                colon_index = 0;
                colon_2_index = 0;
                semicolon_index = 0; )//clean up wrong spaces
                while (InCode.First().Substring(wrong_spaces, 1) == " " && wrong_spaces < InCode.First().Length
                    wrong_spaces++;
                if (InCode.First().Substring(wrong_spaces, 7) == "END_VAR")//end current variable block;
                {
                    result.success = true;
                    break;
                }
                else//here is the process to read variable information
                {//comments "(* comment *)" removal and adding
                    var_temp.comment = RemoveComment(InCode.First()).comment; //write variable's comment
                    TempCodeComment = RemoveComment(InCode.First());
                    if (TempCodeComment.success == false)//check for malformed comments
                    {
                        result.success = false;
                        result.message += "Error in comments at line " + (current_line + result.lines_forward + 1) + ". ";
                        return result;
                    }
                    InCode.RemoveFirst();));//code line clean from comments and empty spaces
                    InCode.AddFirst(RemoveSpaces(TempCodeComment.code
                    if (!(InCode.First().Contains(":") && InCode.First().Contains(";")))//case the declaration line is invalid
                    {
                        result.success = false;
                        result.message += "Error: invalid declaration at line  " + (current_line + result.lines_forward + 1) +
". ";                   return result;
                    }//the declaration line must have all of these elements in order to be valid
                    colon_index = InCode.First().IndexOf(":");
                    semicolon_index = InCode.First().IndexOf(";");
                    //check for consistency of elements' order in declaration
                    if (InCode.First().Contains(":="))//two colons, means we have an initial value declared
                    {
                        colon_2_index = InCode.First().IndexOf(":=");
                        if (colon_2_index > semicolon_index || colon_index > colon_2_index || colon_index >
semicolon_index)//smaller index protection against EOL error;
                        {   result.success = false;
                            result.message += "Error: invalid declaration at line  " + (current_line + result.lines_forward +
1) + ". ";                  return result;}
                    }
                    else
                    {   if (colon_index > semicolon_index)//smaller index protection against EOL error;
                        {   result.success = false;
                            result.message += "Error: invalid declaration at line  " + (current_line + result.lines_forward +
1) + ". ";                  return result;}
                    }//all is good, just read the values
                    var_temp.name = ReplaceIllegalChars(InCode.First().Substring(0, colon_index));
                    if (colon_2_index > 0)
                    {
                        var_temp.type = InCode.First().Substring(colon_index + 1, colon_2_index - colon_index - 1);
                        var_temp.value = InCode.First().Substring(colon_2_index + 2, semicolon_index - colon_2_index - 2);
                        if (var_temp.type == "BOOL")//write initial value
                        {                           //Boolean values conversion from TRUE, FALSE to 1, 0;
                            if (var_temp.value == "TRUE")
                                var_temp.value = "1";
                            if (var_temp.value == "FALSE")
                                var_temp.value = "0";
                        }
                        var_temp.IO_type = IO_type;//write the variable's IO type
                        result.m_file.variables.AddLast(var_temp);
                    }
                    else//no initial value declared
                    {
                        var_temp.type = InCode.First().Substring(colon_index + 1, semicolon_index - colon_index - 1);
                        string default_value = "0"; //aplicable for most var types, just check the exceptions
                        if (var_temp.type=="TIME")
                        {
                            var_temp.type = "INT";}
                        if (var_temp.type == "CTU" || var_temp.type == "CTD" || var_temp.type == "TON")
                        {//FB variable, declare vector of variables required for FB call to work
                            switch (var_temp.type)
                            {
                                case "TON":
                                    TIMER_ON_flag = true;
                                    var_temp_vector = new class_types.m_variable();
                                    var_temp_vector.IO_type = IO_type;
                                    var_temp_vector.retain = false;
                                    var_temp_vector.comment = "";
                                    var_temp_vector.type = "INT";
                                    var_temp_vector.value = "[1,0,0,0,0,0]";
                                    var_temp_vector.name = var_temp.name;
                                    result.m_file.variables.AddLast(var_temp_vector);
                                    var_temp_vector.comment = "FB identifier";
                                    var_temp_vector.type = "STRING";
                                    var_temp_vector.value = "'TIMER_ON'";
                                    var_temp_vector.name = var_temp.name + "_FBID";
                                    result.m_file.variables.AddLast(var_temp_vector);
                                    break;
                                case "CTU":
```

```
                                COUNTER_UP_flag = true;
                                var_temp_vector = new class_types.m_variable();
                                var_temp_vector.IO_type = IO_type;
                                var_temp_vector.retain = false;
                                var_temp_vector.comment = "";
                                var_temp_vector.type = "INT";
                                var_temp_vector.value = "[0,0,0,0,0,0]";
                                var_temp_vector.name = var_temp.name;
                                result.m_file.variables.AddLast(var_temp_vector);
                                var_temp_vector.comment = "FB identifier";
                                var_temp_vector.type = "STRING";
                                var_temp_vector.value = "'COUNTER_UP'";
                                var_temp_vector.name = var_temp.name + "_FBID";
                                result.m_file.variables.AddLast(var_temp_vector);
                                break;
                            case "CTD":
                                COUNTER_DOWN_flag = true;
                                var_temp_vector = new class_types.m_variable();
                                var_temp_vector.IO_type = IO_type;
                                var_temp_vector.retain = false;
                                var_temp_vector.comment = "";
                                var_temp_vector.type = "INT";
                                var_temp_vector.value = "[0,0,0,0,0,0]";
                                var_temp_vector.name = var_temp.name;
                                result.m_file.variables.AddLast(var_temp_vector);
                                var_temp_vector.comment = "FB identifier";
                                var_temp_vector.type = "STRING";
                                var_temp_vector.value = "'COUNTER_DOWN'";
                                var_temp_vector.name = var_temp.name + "_FBID";
                                result.m_file.variables.AddLast(var_temp_vector);
                                break;
                        }
                    }
                    else
                    {//for every simple type, there is a different defaul initial value
                        switch (var_temp.type)
                        {
                            case "BOOL"://TODO ADD THE REMAINING VARIABLE TYPE'S DEFAULT VALUE
                                default_value = "0";
                                break;
                            case "STRING":
                                default_value = "''";
                                break;
                            case "CHAR":
                                default_value = "''";
                                break;
                            case "WORD":
                                default_value = "0";
                                break;
                            case "INT":
                                default_value = "0";
                                break;
                        }
                        var_temp.value=default_value;
                        var_temp.IO_type = IO_type; //write the variable's IO type
                        result.m_file.variables.AddLast(var_temp);
                    }
                }//check for approved variable types
                if (!(var_temp.type == "TON" || var_temp.type == "CTD" || var_temp.type == "CTU" || var_temp.type == "WORD"
|| var_temp.type == "BOOL" || var_temp.type == "BYTE" || var_temp.type == "LWORD" || var_temp.type == "DWORD" || var_temp.type
== "SINT" || var_temp.type == "INT" || var_temp.type == "DINT" || var_temp.type == "LINT" || var_temp.type == "USINT" ||
var_temp.type == "UINT" || var_temp.type == "UDINT" || var_temp.type == "ULINT" || var_temp.type == "REAL" || var_temp.type ==
"LREAL" || var_temp.type == "DATE" || var_temp.type == "TOD" || var_temp.type == "DT" || var_temp.type == "TIME" ||
var_temp.type == "STRING" || var_temp.type == "WSTRING"))
                {   result.success = false;
                    result.message = result.message + "Error: invalid variable type declaration at line  " + (current_line
+ result.lines_forward + 1) + ". ";
                    return result;}
                result.success = true;
                InCode.RemoveFirst();//onto next line
                result.lines_forward++;
            }
        }
        while (InCode.First() == "")//account for empty lines
        {
            InCode.RemoveFirst();
            result.lines_forward++;
            if (InCode.Count <= 0) //in case the EOF is encountered while searching for non-empty lines
            {   result.success = false;
                result.message += "Sudden end of file at line " + (current_line + result.lines_forward + 1) + ". ";
                return result;}
        }
        wrong_spaces = 0; //clean up wrong spaces
        while (InCode.First().Substring(wrong_spaces, 1) == " " && wrong_spaces < InCode.First().Length)
            wrong_spaces++;//reached END_VAR instruction
        if (InCode.First().Substring(wrong_spaces, 7) == "END_VAR")
        {   InCode.RemoveFirst();//onto next line, END_VAR marks the success of the variable declaration
            result.lines_forward++;
            result.success = true;}
        else
        {   result.success = false;
            result.message += "Error: 'END_VAR' expected  at " + (current_line + result.lines_forward + 1) + ". ";}
        return result;}
```

Anexo VI

Listagem do código fonte do método *ReadCode*

```
        class_types.read_function_output ReadCode(LinkedList<string> InCode, class_types.m_file_data m_file_input, int
current_line)// function to read and convert the code
        {
            class_types.read_function_output result;
            class_types.CodeResult TempCodeComment;
            int line_number = 0;
            result.lines_forward = 0;
            result.success = true;
            result.message = "";
            class_types.m_CodeLine workline;
            workline.code="";
            workline.comment="";
            string acumulator = "";
            bool instruction_ran=false;
            class_types.Instruction_and_Argument CodeAndArg;
            while (InCode.Contains(""))//Remove empty lines
                InCode.Remove("");
            string[] TempArray=new string[InCode.Count];
            TempArray = InCode.ToArray();
            result.m_file = m_file_input;
            line_number = 0;

            while (line_number < TempArray.Length) //here the first instruction is expected
            {//if no valid instruction is found, return error
                if (TempArray[line_number].Length > 0)
                {
                    //comments "(* comment *)" removal and adding to var block comment
                    TempCodeComment.code = "";
                    TempCodeComment.comment = "";
                    TempCodeComment.success = false;
                    TempCodeComment = RemoveComment(TempArray[line_number]);
                    if (TempCodeComment.success == false)//check for malformed comments
                    {
                        result.success = false;
                        result.message += "Error in comments at line " + (current_line + line_number + 1) + ". ";
return result;
                    }
                    if (TempCodeComment.comment != "")//write comment to code line
                        workline.comment += TempCodeComment.comment + " || ";
                    TempArray[line_number] = TempCodeComment.code;
                    if (TempArray[line_number] == "")//case of no code and only comment
                    {
                        workline.code = "";
                        result.m_file.codelines.AddLast(workline);
                        workline.comment = "";
                    }
                }
                if (TempArray[line_number].Length < 1 || RemoveSpaces(TempArray[line_number]) == ";")
                    {
                        result.lines_forward++;
                        instruction_ran = true;
                        InCode.RemoveFirst();
                        TempArray = InCode.ToArray();
                    }
                    else
                    {
                        if (TempArray[line_number].Length >= 11 && TempArray[line_number].Substring(0, 11) == "END_PROGRAM")
//END_PROGRAM line
                        {
                            if (result.m_file.codelines.Count <= 0)
                            {
                                result.success = false;
                                result.message += "Empty program. ";
                            }
                            else
                            {
                                result.success = true;
                                result.lines_forward = line_number;
                                for (int j = 0; j < result.lines_forward; j++)
                                    InCode.RemoveFirst();
                                return result;
                            }
                        }

                        try
                        {
                            int index_of_colon = 0;
                            index_of_colon = TempArray[line_number].IndexOf(":");
                            if (index_of_colon > 0 && !TempArray[line_number].Contains(":=")) //LABEL statement, produces 4
lines of code, calling goto.m
                            {
                                GOTO_flag = true;
                                workline.code = "";
                                if (workline.comment != "")
                                {
                                    workline.comment = TempArray[line_number].Substring(0, TempArray[line_number].IndexOf(':'))
+ " LABEL'S COMMENT: ";
                                    result.m_file.codelines.AddLast(workline);
                                }
```

```
                            workline.code = "";
                            workline.comment = "LABEL " +  ReplaceIllegalChars(TempArray[line_number].Substring(0,
index_of_colon));
                            result.m_file.codelines.AddLast(workline);
                            workline.code = "";
                            workline.comment = "";
                            if (TempArray[line_number].Substring(index_of_colon) != "")
                            {
                                if (RemoveSpaces(TempArray[line_number].Substring(index_of_colon)).Length==1 &&
RemoveSpaces(TempArray[line_number].Substring(index_of_colon))==";")
                                {
                                    InCode.RemoveFirst();
                                    TempArray = InCode.ToArray();
                                    result.lines_forward++;
                                }
                                else
                                    TempArray[line_number] = TempArray[line_number].Substring(index_of_colon+1);
                            }
                            else
                            {
                                InCode.RemoveFirst();
                                TempArray = InCode.ToArray();
                                result.lines_forward++;
                            }
                        }
                    }
                    catch { }
                    CodeAndArg = SeparateInstFromArg(TempArray[line_number]);//create a code and argument reference, only
if not END_PROGRAM or LABEL
                    if (CodeAndArg.argument.Contains(".") && CodeAndArg.instruction != "CAL")//case of writing to special
structured variable, such as CTU or TON
                    {   class_types.CodeResult clear_argument=ForwardToFBVar(CodeAndArg.argument,result.m_file);
                        if (clear_argument.success)
                        {
                            CodeAndArg.argument=clear_argument.code;
                            workline.comment+=clear_argument.comment;
                        }
                    }
                    if (CodeAndArg.instruction == "EQ" || CodeAndArg.instruction == "GT" || CodeAndArg.instruction == "GE"
|| CodeAndArg.instruction == "NE" || CodeAndArg.instruction == "LE" || CodeAndArg.instruction == "LT")
                    {
                        string operation = "";
                        switch (CodeAndArg.instruction)
                        {
                            case "EQ":
                                operation = "==";
                                break;
                            case "GT":
                                operation = ">";
                                break;
                            case "GE":
                                operation = ">=";
                                break;
                            case "NE":
                                operation = "~=";
                                break;
                            case "LE":
                                operation = "<=";
                                break;
                            case "LT":
                                operation = "<";
                                break;
                        }
                        acumulator = "(" + acumulator + operation + " " + ReplaceIllegalChars(CodeAndArg.argument) + ")";
                    }
                    if (CodeAndArg.instruction.Length>2 && CodeAndArg.instruction.Substring(0, 3) == "RET")
                    {
                        workline.code = "return";
                        result.m_file.codelines.AddLast(workline);
                        workline.comment = "";
                        instruction_ran = true;
                        result.lines_forward++;
                    }
                    if (CodeAndArg.instruction == "CAL")//identify the type of call
                    {
                        string FB_ID="";
                        string name = "";
                        if (CodeAndArg.argument.Contains("("))
                        {
                            for (int i = 0; i < result.m_file.variables.Count; i++)//find variable
                            {
                                if (CodeAndArg.argument.Substring(0, CodeAndArg.argument.IndexOf("(")) + "_FBID" ==
result.m_file.variables.ElementAt(i).name)
                                {
                                    FB_ID = result.m_file.variables.ElementAt(i).value;//retrieve FB_ID
                                    name = result.m_file.variables.ElementAt(i).name.Substring(0,
result.m_file.variables.ElementAt(i).name.IndexOf("_FBID"));
                                    break;
                                }
                            }
                        }
                        if (FB_ID != "")
                        {   LinkedList<string> identifiers = new LinkedList<string>();
                            LinkedList<class_types.CodeResult> parameter_vars = new LinkedList<class_types.CodeResult>();
                            switch (FB_ID)
                            {
                                case "'TIMER_ON'":
                                    TIMER_ON_flag = true;
                                    workline.code = "";
                                    if (workline.comment != "")
                                    {
```

```
                                        workline.comment = name + " FB call comment: " + workline.comment;
                                        result.m_file.codelines.AddLast(workline);
                                    }//read parameters
                                    identifiers.Clear();//format IN:=VAR1,PT:= VAR2,CV=>VAROUT
                                    identifiers.AddLast("IN:=");
                                    identifiers.AddLast("PT:=");
                                    identifiers.AddLast("ET=>");
                                    parameter_vars.Clear();
                                    parameter_vars = ReadParametersFromIdentifier(CodeAndArg.argument, identifiers,
result.m_file);
                                    workline.comment = "";
                                    if (parameter_vars.Count > 0 && parameter_vars.ElementAt(0).success)
                                    {
                                        workline.code += name + "(2)=" + parameter_vars.ElementAt(0).code + ";";
if (parameter_vars.ElementAt(0).comment != "")
                                            workline.comment += parameter_vars.ElementAt(0).comment;
                                    }
                                    if (parameter_vars.Count > 1 && parameter_vars.ElementAt(1).success)
                                    {
                                        workline.code += name + "(4)=" + parameter_vars.ElementAt(1).code + ";";
if (parameter_vars.ElementAt(1).comment != "")
                                            workline.comment += parameter_vars.ElementAt(0).comment;
                                    }
                                    workline.code += name + "=TIMER_ON(CLOCK_IN," + name + ");";
                                    if (parameter_vars.Count>2 && parameter_vars.ElementAt(2).success &&
parameter_vars.ElementAt(2).code != "0")//write CV value to designated variable
                                        workline.code += parameter_vars.ElementAt(2).code + "=" + name + "(3)";
result.m_file.codelines.AddLast(workline);
                                    break;
                                case "'COUNTER_DOWN'":
                                    COUNTER_DOWN_flag = true;
                                    workline.code = "";
                                    if (workline.comment != "")
                                    {
                                        workline.comment = name + " FB call comment: " + workline.comment;
result.m_file.codelines.AddLast(workline);
                                    }//read parameters
                                    identifiers.Clear();
                                    identifiers.AddLast("LOAD:=");
                                    identifiers.AddLast("PV:=");
                                    identifiers.AddLast("CV=>");
                                    parameter_vars.Clear();
                                    parameter_vars = ReadParametersFromIdentifier(CodeAndArg.argument, identifiers,
result.m_file);
                                    workline.comment = "";
                                    if (parameter_vars.Count > 0 && parameter_vars.ElementAt(0).success)
                                    {
                                        workline.code += name + "(5)=" + parameter_vars.ElementAt(0).code + ";";
if (parameter_vars.ElementAt(0).comment != "")
                                            workline.comment += parameter_vars.ElementAt(0).comment;
                                    }
                                    if (parameter_vars.Count > 1 && parameter_vars.ElementAt(1).success)
                                    {
                                        workline.code += name + "(4)=" + parameter_vars.ElementAt(1).code + ";";
if (parameter_vars.ElementAt(1).comment != "")
                                            workline.comment += parameter_vars.ElementAt(0).comment;
                                    }
                                    workline.code += name + "=COUNTER_DOWN(" + name + ");";
                                    if (parameter_vars.Count>2 && parameter_vars.ElementAt(2).success &&
parameter_vars.ElementAt(2).code != "0")//write CV value to designated variable
                                        workline.code += parameter_vars.ElementAt(2).code + "=" + name + "(3)";
result.m_file.codelines.AddLast(workline);
                                    workline.comment = "";
                                    break;
                                case "'COUNTER_UP'":
                                    COUNTER_UP_flag = true;
                                    workline.code = "";
                                    if (workline.comment != "")
                                    {
                                        workline.comment = name + " FB call comment: " + workline.comment;
result.m_file.codelines.AddLast(workline);
                                        workline.comment = "";
                                    }
                                    identifiers.Clear();//read parameters
                                    identifiers.AddLast("RESET:=");
                                    identifiers.AddLast("PV:=");
                                    identifiers.AddLast("CV=>");
                                    parameter_vars.Clear();
                                    parameter_vars = ReadParametersFromIdentifier(CodeAndArg.argument, identifiers,
result.m_file);            workline.comment = "";
                                    if (parameter_vars.Count > 0 && parameter_vars.ElementAt(0).success)
                                    {
                                        workline.code += name + "(5)=" + parameter_vars.ElementAt(0).code + ";";
if (parameter_vars.ElementAt(0).comment != "")
                                            workline.comment += parameter_vars.ElementAt(0).comment;
                                    }
                                    if (parameter_vars.Count > 1 && parameter_vars.ElementAt(1).success)
                                    {
                                        workline.code += name + "(4)=" + parameter_vars.ElementAt(1).code + ";";
if (parameter_vars.ElementAt(1).comment != "")
                                            workline.comment += parameter_vars.ElementAt(0).comment;
                                    }
                                    workline.code += name + "=COUNTER_UP(" + name + ");";
                                    if (parameter_vars.Count>2 && parameter_vars.ElementAt(2).success &&
parameter_vars.ElementAt(2).code != "0")//write CV value to designated variable
                                        workline.code += parameter_vars.ElementAt(2).code + "=" + name + "(3);";
result.m_file.codelines.AddLast(workline);
                                    workline.comment = "";
                                    break;
                            }
                        }
```

```
                            else
                            {
                                workline.code = CodeAndArg.argument.Substring(0,CodeAndArg.argument.IndexOf("(") ) + "(" +
CodeAndArg.argument.Substring(CodeAndArg.argument.IndexOf("(") + 1, CodeAndArg.argument.IndexOf(")") -
CodeAndArg.argument.IndexOf("(") - 1) + ");";
                                result.m_file.codelines.AddLast(workline);
                                workline.comment = "";
                                instruction_ran = true;
                            }
                        }
                        if (CodeAndArg.instruction == "JMPC" || CodeAndArg.instruction == "JMPCN" || CodeAndArg.instruction ==
"JMP")//JMP statements
                        {
                            GOTO_flag = true;
                            if (acumulator == "")
                            {
                                result.success = false;
                                result.message += "Error in code at line " + (current_line + line_number + 1) + ": JMPC
instruction before any value is in the acumulator. ";
                                return result;
                            }
                            if (CodeAndArg.instruction == "JMP")
                            {
                                workline.comment = "";
                                workline.code = "goto('" + ReplaceIllegalChars(CodeAndArg.argument) + "');";
                                result.m_file.codelines.AddLast(workline);
                                workline.code = "return";
                                workline.comment = "Part of the goto functionality in Matlab, ignore";
                                result.m_file.codelines.AddLast(workline);
                            }
                            else
                            {
                                if (CodeAndArg.instruction == "JMPCN")
                                    workline.code = "if (" + acumulator + "==0)";
                                else
                                    workline.code = "if (" + acumulator + "==1)";
                                result.m_file.codelines.AddLast(workline);
                                workline.comment = "";
                                workline.code = "    goto('" + ReplaceIllegalChars(CodeAndArg.argument) + "');";
result.m_file.codelines.AddLast(workline);
                                workline.code = "    return";
                                workline.comment = "Part of the goto functionality in Matlab, ignore";
                                result.m_file.codelines.AddLast(workline);
                                workline.comment = "";
                                workline.code = "end";
                                result.m_file.codelines.AddLast(workline);
                                workline.comment = "";
                                instruction_ran = true;
                            }
                        }
                        if (CodeAndArg.instruction == "ST" || CodeAndArg.instruction == "ST(") //ST instruction found, begin
backtracking from ElementAt(end_instruction_line) up to First()
                        {
                            if (acumulator == "")//In case ST is the first instruction encountered
                            {
                                result.success = false;
                                result.message += "Error in code at line " + (current_line + line_number + 1) + ": ST found
before any LD.";
                                return result;
                            }//start writing the workline
                            workline.code = ReplaceIllegalChars(RemoveEndSpaces(CodeAndArg.argument)) + " = " + acumulator +
";";//If a acumulator like behaviour is needed, we use this. Examples: JMPC, EQ, etc..
                            result.m_file.codelines.AddLast(workline);
                            workline.comment = "";
                            workline.code = "";
                            result.lines_forward++;
                        }
                        if (CodeAndArg.instruction == "S" || CodeAndArg.instruction == "R")
                        {
                            workline.code = ReplaceIllegalChars(CodeAndArg.argument) + " = ";
                            if (CodeAndArg.instruction == "S")
                                workline.code += 1+";";
                            if (CodeAndArg.instruction == "R")
                                workline.code += 0+";";
                            result.m_file.codelines.AddLast(workline);
                            workline.comment = "";
                            workline.code = "";
                            instruction_ran = true;
                            result.lines_forward++;
                        }
                        switch (CodeAndArg.instruction)//behaviour dependent on the instruction
                        {
                            case ")":
                                acumulator += ")";
                                instruction_ran = true;
                                break;
                            case "LD":
                                acumulator = ReplaceIllegalChars(CodeAndArg.argument);
                                instruction_ran = true;
                                break;
                            case "LDN":
                                acumulator = "~" + ReplaceIllegalChars(CodeAndArg.argument);
                                instruction_ran = true;
                                break;
                            case "LD(":
                                acumulator = "(" + ReplaceIllegalChars(CodeAndArg.argument);
                                instruction_ran = true;
                                break;
                            case "LDN(":
                                acumulator = "(~" + ReplaceIllegalChars(CodeAndArg.argument);
                                instruction_ran = true;
```

```
                        break;
                    case "AND":
                        acumulator = "(" + acumulator;
                        acumulator += "&" + ReplaceIllegalChars(CodeAndArg.argument) + ")";
                        instruction_ran = true;
                        break;
                    case "AND(":
                        acumulator = "((" + acumulator;
                        acumulator += "&" + ReplaceIllegalChars(CodeAndArg.argument) + ")";
                        instruction_ran = true;
                        break;
                    case "&":
                        acumulator = "(" + acumulator;
                        acumulator += "&" + ReplaceIllegalChars(CodeAndArg.argument) + ")";
                        instruction_ran = true;
                        break;
                    case "&(":
                        acumulator = "((" + acumulator;
                        acumulator += "&" + ReplaceIllegalChars(CodeAndArg.argument) + ")";
                        instruction_ran = true;
                        break;
                    case "&~":
                        acumulator = "(" + acumulator;
                        acumulator += "&~" + ReplaceIllegalChars(CodeAndArg.argument) + ")";
                        instruction_ran = true;
                        break;
                    case "ANDN":
                        acumulator = "(" + acumulator;
                        acumulator += "&~" + ReplaceIllegalChars(CodeAndArg.argument) + ")";
                        instruction_ran = true;
                        break;
                    case "ORN":
                        acumulator = "(" + acumulator;
                        acumulator += "|~" + ReplaceIllegalChars(CodeAndArg.argument) + ")";
                        instruction_ran = true;
                        break;
                    case "ORN(":
                        acumulator = "(" + acumulator;
                        acumulator += "|~(" + ReplaceIllegalChars(CodeAndArg.argument) + ")";
                        instruction_ran = true;
                        break;
                    case "OR":
                        acumulator = "(" + acumulator;
                        acumulator += "|" + ReplaceIllegalChars(CodeAndArg.argument) + ")";
                        instruction_ran = true;
                        break;
                    case "OR(":
                        acumulator += "|(" + ReplaceIllegalChars(CodeAndArg.argument);
                        instruction_ran = true;
                        break;
                    case "XOR":
                        acumulator = "xor(" + acumulator + "," + ReplaceIllegalChars(CodeAndArg.argument) + ")";
                        instruction_ran = true;
                        break;
                    case "XORN":
                        acumulator = "xor(" + acumulator + "," + ReplaceIllegalChars(CodeAndArg.argument) + ")";
                        instruction_ran = true;
                        break;
                    case "ADD":
                        acumulator = "(" + acumulator;
                        acumulator += "+" + ReplaceIllegalChars(CodeAndArg.argument) + ")";
                        instruction_ran = true;
                        break;
                    case "ADD(":
                        acumulator = "((" + acumulator;
                        acumulator += "+" + ReplaceIllegalChars(CodeAndArg.argument) + ")";
                        instruction_ran = true;
                        break;
                    case "SUB":
                        acumulator = "(" + acumulator;
                        acumulator += "-" + ReplaceIllegalChars(CodeAndArg.argument) + ")";
                        instruction_ran = true;
                        break;
                    case "DIV":
                        acumulator = "(" + acumulator;
                        acumulator += "/" + ReplaceIllegalChars(CodeAndArg.argument) + ")";
                        instruction_ran = true;
                        break;
                    case "MUL":
                        acumulator = "(" + acumulator;
                        acumulator += "*" + ReplaceIllegalChars(CodeAndArg.argument) + ")";
                        instruction_ran = true;
                        break;
                }
                if (instruction_ran)
                {
                    InCode.RemoveFirst();
                    TempArray = InCode.ToArray();
                }
            }
        }
            result.lines_forward = line_number;
            for (int i = 0; i < result.lines_forward; i++)//remove the processed lines
                InCode.RemoveFirst();
            result.success = false;
            result.message += "No END_PROGRAM instruction found. ";
            return result;
    }
```

Anexo VII

Listagem do código de implementação em *Matlab* do POU TON

```
%Timer, iterable

function [TIMER_OUTPUT]=TIMER_ON(CLOCK_IN,TIMER_INPUT)
%INPUT FORMAT: IN_OLD,IN,ET,PT,TARGET_TIME_ABSOLUTE,Q


if TIMER_INPUT(1)==0 & TIMER_INPUT(2)==1
    TIMER_INPUT(6)=0;
    TIMER_INPUT(3)=0;
    TIMER_INPUT(5)=CLOCK_IN+TIMER_INPUT(4);
end

if TIMER_INPUT(1)==1 & TIMER_INPUT(2)==0
    TIMER_INPUT(3)=0;
    TIMER_INPUT(6)=0;
    TIMER_INPUT(5)=0;
end

%Update elapsed tim
if TIMER_INPUT(2)==1
    TIMER_INPUT(3)=CLOCK_IN+TIMER_INPUT(4)-TIMER_INPUT(5);
end

if TIMER_INPUT(4)>0 & (CLOCK_IN>=TIMER_INPUT(5)) & TIMER_INPUT(2)==1
   TIMER_INPUT(6)=1;
else
   TIMER_INPUT(6)=0;
end


%Update diferential values
TIMER_INPUT(1)=TIMER_INPUT(2);

TIMER_OUTPUT=TIMER_INPUT;
%OUTPUT FORMAT: IN_OLD,IN,ET,PT,START_TIME_ABSOLUTE,Q
```

Anexo VIII

Listagem do código de implementação em *Matlab* do POU CTU

```
%Up Counter, iterable

function [COUNTER_UP_OUTPUT]=COUNTER_UP(COUNTER_UP_INPUT)
%INPUT FORMAT
(COUNTER_UP_CU_OLD,COUNTER_UP_CU,COUNTER_UP_CV,COUNTER_UP_PV,
COUNTER_UP_RESET,COUNTER_UP_Q)

%Counter reset
if COUNTER_UP_INPUT(5)==1
    COUNTER_UP_INPUT(6)=0;
    COUNTER_UP_INPUT(3)=0;
    COUNTER_UP_INPUT(4)=0;
    COUNTER_UP_INPUT(1)=COUNTER_UP_INPUT(2);
else

    %Count up

    if COUNTER_UP_INPUT(2)==1 & COUNTER_UP_INPUT(1)==0
        COUNTER_UP_INPUT(3)=COUNTER_UP_INPUT(3)+1;
    end

    %Predefined value reached

    if COUNTER_UP_INPUT(3)>=COUNTER_UP_INPUT(4)
        COUNTER_UP_INPUT(6)=1;
    else
        COUNTER_UP_INPUT(6)=0;
    end
end

%Update diferential values

COUNTER_UP_INPUT(1)=COUNTER_UP_INPUT(2);

%Write ouput vector

COUNTER_UP_OUTPUT=COUNTER_UP_INPUT;
%OUTPUT FORMAT
(COUNTER_UP_CU_OLD,COUNTER_UP_CU,COUNTER_UP_CV,COUNTER_UP_PV,
COUNTER_UP_RESET,COUNTER_UP_Q)
```

Anexo IX

Listagem do código de implementação em *Matlab* do POU CTD

```
%Down Counter, iterable

function [COUNTER_DOWN_OUTPUT]=COUNTER_DOWN(COUNTER_DOWN_INPUT)
%INPUT FORMAT
(COUNTER_DOWN_CD_OLD,COUNTER_DOWN_CD,COUNTER_DOWN_INPUT(3),
COUNTER_DOWN_INPUT(4),COUNTER_UP_LOAD,COUNTER_UP_Q)

%Counter load
if COUNTER_DOWN_INPUT(5)==1
    COUNTER_DOWN_INPUT(6)=0;
    COUNTER_DOWN_INPUT(3)=COUNTER_DOWN_INPUT(4);
    COUNTER_DOWN_INPUT(1)=COUNTER_DOWN_INPUT(2);
else

    %Count down

    if COUNTER_DOWN_INPUT(2)==1 & COUNTER_DOWN_INPUT(1)==0
        COUNTER_DOWN_INPUT(3)=COUNTER_DOWN_INPUT(3)-1;
    end

    %Predefined value reached

    if COUNTER_DOWN_INPUT(3)<=0
        COUNTER_DOWN_INPUT(6)=1;
    else
        COUNTER_DOWN_INPUT(6)=0;
    end
end

%Update diferential values

COUNTER_DOWN_INPUT(1)=COUNTER_DOWN_INPUT(2);

%Write ouput vector

COUNTER_DOWN_OUTPUT=COUNTER_DOWN_INPUT;
%OUTPUT FORMAT
(COUNTER_DOWN_CD_OLD,COUNTER_DOWN_CD,COUNTER_DOWN_INPUT(3),
COUNTER_DOWN_INPUT(4),COUNTER_UP_LOAD,COUNTER_UP_Q)
```

Anexo X

Listagem do código fonte do método *ApplyGeneralRules*

```
        private void ApplyGeneralRules(XmlDocument Dictionary)
        {
            class_types.IL_CodeLine codeline = new class_types.IL_CodeLine();

            XmlNodeList Rules =
Dictionary.SelectNodes("//conversion_library//conversion_rules//general_rules//general_rule//outcome");
            string outcome_instruction = "";

            if (Rules != null)
            {
                foreach (XmlNode node in Rules)
                {
                    //outcome_instruction = GetXmlNodeValueByTag(node, "outcome");
                    outcome_instruction = node.InnerText;

                        //literal
                        if (System.Text.Encoding.UTF8.GetBytes(outcome_instruction)[0] == 34 &&
System.Text.Encoding.UTF8.GetBytes(outcome_instruction)[System.Text.Encoding.UTF8.GetBytes(outcome_instruction).Length - 1] ==
34)
                        {
                            outcome_instruction = outcome_instruction.Substring(1);
                            outcome_instruction = outcome_instruction.Substring(0, outcome_instruction.Length - 1);

                            //-----------Remove comments------------------------------
                            string comment_marker_start =
Dictionary.SelectSingleNode("//conversion_library//conversion_rules//comment_markers//start").InnerText;
                            string comment_marker_end =
Dictionary.SelectSingleNode("//conversion_library//conversion_rules//comment_markers//end").InnerText;
                            class_types.CodeResult comments_operation = RemoveComment(outcome_instruction,
comment_marker_start, comment_marker_end);

                            codeline.comment = comments_operation.comment;
                            outcome_instruction = comments_operation.code;

                            //-------------End remove comments------------------------------


                            IL_file.codelines.AddLast(codeline);

                        }
                }
            }
        }
```

Anexo XI

Listagem do código fonte do método *CheckForInstructions*

```
private bool CheckForInstructions( int CurrentLine, XmlDocument Dictionary)
        {
            XmlNodeList instructions =
Dictionary.SelectNodes("//conversion_library//conversion_rules//instructions//instruction//old_instruction");
            class_types.Instruction_and_Argument old_code,new_code;
            class_types.IL_CodeLine codeline = new class_types.IL_CodeLine();
            class_types.CodeResult comments_operation = new class_types.CodeResult();
            string comment_marker_start
=Dictionary.SelectSingleNode("//conversion_library//conversion_rules//comment_markers//start").InnerText;
            string comment_marker_end =
Dictionary.SelectSingleNode("//conversion_library//conversion_rules//comment_markers//end").InnerText;
            bool success=true;
            new_code.argument = "";
            new_code.instruction = "";
            comments_operation = RemoveComment(InCode.ElementAt(CurrentLine), comment_marker_start, comment_marker_end);
            if (comments_operation.code == "")//check for comments only line
            {   codeline.code = "";
                codeline.comment = comments_operation.comment;
                IL_file.codelines.AddLast(codeline);}
            else //line with code
            {
                for (int i = 0; i < instructions.Count; i++)//-----------Remove comments----------
                {
                    comments_operation = RemoveComment(InCode.ElementAt(CurrentLine), comment_marker_start,
comment_marker_end);if (comments_operation.success == false)
                    {   success = false;
                        break;}
                    old_code = SeparateInstFromArg(comments_operation.code); //old now has the old line
                    XmlNodeList current_instruction = instructions.Item(i).ParentNode.ChildNodes;
                    if (CompareWithIgnores(instructions.Item(i).InnerText,old_code.instruction))
                    {
                        codeline.code = "";
                        codeline.comment = comments_operation.comment;//---OUTCOME WRITER----
                        new_code = SeparateInstFromArg(GetXmlNodeValueByTag(current_instruction, "outcome"));
                        while (new_code.instruction != "")//check for literal i.e. "
                        {
                            if (System.Text.Encoding.UTF8.GetBytes(new_code.instruction)[0] == 34 &&
System.Text.Encoding.UTF8.GetBytes(new_code.instruction)[System.Text.Encoding.UTF8.GetBytes(new_code.instruction).Length - 1]
== 34)                            {
                                new_code.instruction = new_code.instruction.Substring(1);
                                new_code.instruction = new_code.instruction.Substring(0, new_code.instruction.Length - 1);
                                codeline.code += new_code.instruction;
                                new_code = SeparateInstFromArg(new_code.argument);
                                }
                            else
                            {
                                switch (new_code.instruction)
                                {
                                    case "instruction":
                                        codeline.code += " " + GetXmlNodeValueByTag(current_instruction, "new_instruction");
                                        new_code = SeparateInstFromArg(new_code.argument);
                                        break;
                                    case "argument":
                                        codeline.code += " " + ReplaceIllegalChars(old_code.argument);
                                        new_code = SeparateInstFromArg(new_code.argument);
                                        break;
                                    case "old_instruction":
                                        codeline.code += " "+ old_code.instruction;
                                        new_code = SeparateInstFromArg(new_code.argument);
                                        break;
                                }
                            }
                        }
                        if ((codeline.code != "" || codeline.comment != "") && new_code.instruction == "")//add written line
                        {
                            if (codeline.code.Length > 0 && codeline.code.Substring(0, 1) == " ")//clear first space
                                codeline.code = codeline.code.Substring(1);
                            IL_file.codelines.AddLast(codeline);
                        }
                        LinkedList<string> rules = new LinkedList<string>();//RULES
                        rules = GetXmlNodeValuesByTag(current_instruction, "rules");
                        for (int j = 0; j < rules.Count; j++)
                        {
                            new_code.argument = "";
                            new_code.instruction = "";
                            new_code = SeparateInstFromArg(rules.ElementAt(j));

                            CheckForRules(CurrentLine, Dictionary, current_instruction, old_code, new_code);
                        }
                    }
                }}
            return success;}
```

Anexo XII

Listagem do código fonte do método *CheckForRules*

```
private void CheckForRules(int CurrentLine, XmlDocument Dictionary, XmlNodeList current_instruction,
class_types.Instruction_and_Argument old_code, class_types.Instruction_and_Argument new_code)
        {
            class_types.IL_CodeLine codeline = new class_types.IL_CodeLine();
            string location = "";
            string reference = "";
            string which = "";
            int index = 0;
            while (new_code.instruction != "")
            {
                switch (new_code.instruction)
                {//----No spaces---
                    case "delete": //delete what where
                        new_code = SeparateInstFromArg(new_code.argument);
                        bool special = false;
                        string to_delete = "";//literal
                        if (System.Text.Encoding.UTF8.GetBytes(new_code.instruction)[0] == 34 &&
System.Text.Encoding.UTF8.GetBytes(new_code.instruction)[System.Text.Encoding.UTF8.GetBytes(new_code.instruction).Length - 1]
== 34)
                        {
                            new_code.instruction = new_code.instruction.Substring(1);
                            new_code.instruction = new_code.instruction.Substring(0, new_code.instruction.Length - 1);
                            to_delete = new_code.instruction;
                        }
                        else
                        {
                            switch (new_code.instruction)
                            {
                                case "space":
                                    to_delete = " ";
                                    break;
                                case "line":
                                    special = true;
                                    to_delete = "line";
                                    break;
                            }
                        }
                        if (!special)
                        {//delete a string element from the place
                            new_code = SeparateInstFromArg(new_code.argument);
                            switch (new_code.instruction)
                            {
                                case "outcome":
                                    codeline = IL_file.codelines.Last();
                                    IL_file.codelines.RemoveLast();
                                    codeline.code = RemoveString(codeline.code, to_delete);
                                    IL_file.codelines.AddLast(codeline);
                                    break;
                                case "argument":
                                    break;
                            }
                        }
                        else
                        {//case of special
                            new_code = SeparateInstFromArg(new_code.argument);
                            //find index
                            location = "";
                            reference = "";
                            which = "";
                            location = new_code.instruction;//location now contains "before", "after" or "at"
                            new_code = SeparateInstFromArg(new_code.argument);
                            which = new_code.instruction;//which now contains "first","last", "line"
                            new_code = SeparateInstFromArg(new_code.argument);
                            reference = new_code.instruction;//reference now contains "new_stack", "..."
                            //case of literal, accepts number of line
                            if (System.Text.Encoding.UTF8.GetBytes(reference)[0] == 34 &&
System.Text.Encoding.UTF8.GetBytes(reference)[System.Text.Encoding.UTF8.GetBytes(reference).Length - 1] == 34)
                            {
                                reference = reference.Substring(1);
                                reference = reference.Substring(0, reference.Length - 1);
                                try
                                {IL_file.codelines = RemoveLinkedListCodeline(IL_file.codelines, int.Parse(reference));}
                                Catch {}
                            }
                            else
                            {
                                index = GetIndexFromOrders(Dictionary, location, which, reference);
                                if (index < 0 && index > IL_file.codelines.Count)
                                    break;
                                IL_file.codelines= RemoveLinkedListCodeline(IL_file.codelines, index);
                            }
                        }
                    break;


                    case "declare":
                        class_types.IL_variable declare_line = new class_types.IL_variable();
                        declare_line.type = null;
                        bool exists = false;
                        new_code = SeparateInstFromArg(new_code.argument);
```

```
                        for (int j = 0; j < IL_file.variables.Count; j++)
                        {
                            if (IL_file.variables.ElementAt(j).name == ReplaceIllegalChars(old_code.argument))
                            {exists = true;
                                break;}
                        }
                        if (!exists && new_code.instruction == "argument")
                        {
                            XmlNodeList var_types =
Dictionary.SelectNodes("//conversion_library//variable_types//variable_type");//run through variable types
                            XmlNodeList var_type_result = CompareVarType(var_types, old_code.argument);
                            class_types.Instruction_and_Argument type_node =
SeparateInstFromArg(GetXmlNodeValueByTag(var_type_result, "new"));
                            string argument = SeparateInstFromArg(type_node.argument).instruction;
                            string remainder =
                            declare_line.name = ReplaceIllegalChars(old_code.argument);
                            declare_line.IO_type = type_node.instruction;
                            if (argument != "")
                            {
                                string type_real = "";
                                if (argument == "binary")
                                    type_real = "BOOL";
                                if (argument == "digital")
                                    type_real = "BOOL";
                                declare_line.type = type_real;
                            }
                            if (SeparateInstFromArg(type_node.argument).argument == "" && declare_line.type != null)
                            {
                                IL_file.variables.AddLast(declare_line);
                                new_code.argument = "";
                                new_code.instruction = "";
                            }
                            else
                                break;
                        }
                        else
                        {new_code.argument = "";
                            new_code.instruction = "";}
                        break;
                    case "add":
                        string simbolic_codeline = "";
                        codeline.code = "";
                        codeline.comment = "";
                        while (true)
                        {
                            new_code = SeparateInstFromArg(new_code.argument);
                            if (new_code.instruction == "" || new_code.instruction == "at" || new_code.instruction == "before"
|| new_code.instruction == "after")
                                break;
                            if (simbolic_codeline == "")
                                simbolic_codeline += new_code.instruction;
                            else
                                simbolic_codeline += " " + new_code.instruction;
                        }//here, the codeline is simbolic
                        location = "";//find index
                        reference = "";
                        which = "";
                        location = new_code.instruction;//location now contains "before", "after" or "at"
                        new_code = SeparateInstFromArg(new_code.argument);
                        which = new_code.instruction;//which now contains "first" or "last"
                        new_code = SeparateInstFromArg(new_code.argument);
                        reference = new_code.instruction;//reference now contains "new_stack", "..."
                        index = GetIndexFromOrders(Dictionary, location, which, reference);
                        if (index < 0 && index > IL_file.codelines.Count)
                            break;
                        while (simbolic_codeline != "")//knowing the index and what to write
                        {// generate the actual codeline from the simbolic one
                            if (codeline.code != "")
                                codeline.code += " ";
                            switch (SeparateInstFromArg(simbolic_codeline).instruction)
                            {
                                case "new_instruction":
                                    codeline.code += GetXmlNodeValueByTag(current_instruction, "new_instruction");
                                    break;
                                case "argument":
                                    codeline.code += SeparateInstFromArg(IL_file.codelines.ElementAt(index).code).argument;
                                    break;
                            }
                            simbolic_codeline = SeparateInstFromArg(simbolic_codeline).argument;
                        }
                        IL_file.codelines = InsertCodeAtIndexOfList(IL_file.codelines, codeline, index);
                        break;
                    default:
                        new_code.instruction = "";
                        break;
                }
            }
        }
```

Anexo XIII

Listagem do código *Q Series* traduzido da experiência de tradução

```
1  PROGRAM Q_Series_test_code (*Original filename: Q Series test code.qs*)
2  (*Translated by IEC 61131-3 IL Translator by Andre' Pereira*)
3  (*02-07-2011 15:28:11 Hora padrão de GMT*)
4  (*Dictionary used: Mitsubishi Q Series List Mode version: 0.2*)
5
6  VAR_GLOBAL
7  M0_0 : BOOL;
8  M0_1 : BOOL;
9  END_VAR
10
11 VAR_INPUT
12 I0_1 : BOOL;
13 I0_5 : BOOL;
14 I0_3 : BOOL;
15 END_VAR
16
17 (* Mitsubishi Q Series translation dictionary *)
18
19 (* START PROGRAM *)
20 LD I0_1
21 JMPC P0
22 LD I0_5
23 AND I0_3
24 OR( M0_0
25 ORN M0_1
26 ) (*  OR BLOCK *)
27 ST M0_0
28 JMP P1
29 P0:
30 LD 0
31 ST M0_0
32 P1:
33
34 END_PROGRAM
```

Anexo XIV

Listagem do código STL traduzido da experiência de tradução

```
1   PROGRAM STL_test_code (*Original filename: STL test code.stl*)
2   (*Translated by IEC 61131-3 IL Translator by Andre' Pereira*)
3   (*02-07-2011 15:28:16 Hora padrão de GMT*)
4   (*Dictionary used: Siemens Step 7 S-200 STL version: 0.5*)
5
6   VAR_GLOBAL
7   M0_0 : BOOL;
8   M0_1 : BOOL;
9   END_VAR
10
11  VAR_INPUT
12  I0_1 : BOOL;
13  I0_5 : BOOL;
14  I0_3 : BOOL;
15  END_VAR
16
17  (* Siemens Step 7 S-200 STL translation dictionary *)
18
19  (* START PROGRAM *)
20  LD I0_1
21  JMPC 0
22  LD I0_5
23  AND I0_3
24  OR( M0_0
25  ORN M0_1
26  ) (* OR BLOCK *)
27  ST M0_0
28  LD 1
29  JMPC 1
30  0:
31  LD 0
32  ST M0_0
33  1:
34
35  END_PROGRAM
```

Anexo XV

Listagem do código PLC em IL de controlo do nível de água no tanque

```
1  PROGRAM WATER_TANK (*Controls the water level of a tank*)
2
3  VAR
4  Timer1:TON;
5  Counter1 : CTU;
6  END_VAR
7
8  VAR_INPUT
9  Maximum_Level_Switch : BOOL;
10 Minimum_Level_Switch : BOOL;
11 External_Counter_Reset:BOOL;
12 END_VAR
13
14 VAR_OUTPUT
15 alarm_counter: int;
16 valve:bool;
17 Light_Signal_Alarm:bool;
18 Buzz_Signal_Alarm:   BooL;
19 END_VAR
20
21 LD valve
22 JMPC VALVEON
23 (*Valve not on branch*)
24 LD Maximum_Level_Switch;
25 JMPCN ALLDONE
26 (*If maximum level is true*)
27 S valve;
28 S Counter1.CU;
29 JMP ALLDONE
30
31 (*If Valve on branch*)
32 VALVEON:
33 LD Minimum_Level_Switch;
34 JMPC ALLDONE;
35 (*If water is below minimum and valve is on*)
36 R valve;
37 R Counter1.CU;
38
39 ALLDONE: (*valve control calculated, onto alarm program*)
40 CAL Counter1(Reset:=External_Counter_Reset,PV:=8,CV=>alarm_counter)
41
42 CAL Timer1(IN:=Counter1.Q,PT:=1,ET=>0)
43
44 LD Timer1.Q
45 ST Buzz_Signal_Alarm;
46
47 LD Counter1.Q;
48 ST Light_Signal_Alarm;
49
50 END_PROGRAM
```

Anexo XVI

Listagem do código de controlo do nível de água no tanque convertido para *Matlab*

```
Function [OUTPUT_VECTOR]=WATER_TANK(CLOCK_IN,EXTERNAL_COUNTER_RESET,
MAXIMUM_LEVEL_SWITCH,MINIMUM_LEVEL_SWITCH)
%CONTROLS THE WATER LEVEL OF A TANK
%Converted by IEC-Matlab converter by Andre' Pereira
%03-07-2011 23:26:38 Hora padrÃ£o de GMT


global ALARM_COUNTER BUZZ_SIGNAL_ALARM COUNTER1 COUNTER1_FBID
LIGHT_SIGNAL_ALARM TIMER1 TIMER1_FBID VALVE

%-------------------------Variables' comments-----------------------
%global COUNTER1_FBID's comment: FB identifier
%global TIMER1_FBID's comment: FB identifier
%-------------------------------------------------------------------

if (VALVE==1)
    goto('VALVEON');
    return % Part of the goto functionality in Matlab, ignore
end
% VALVE NOT ON BRANCH ||
if (MAXIMUM_LEVEL_SWITCH==0)
    goto('ALLDONE');
    return % Part of the goto functionality in Matlab, ignore
end
% IF MAXIMUM LEVEL IS TRUE ||
VALVE = 1;
COUNTER1(2) = 1; % COUNTER1.CU
goto('ALLDONE');
return % Part of the goto functionality in Matlab, ignore
% Part of the goto functionality in Matlab, ignoreIF VALVE ON BRANCH
||
% LABEL VALVEON
if (MINIMUM_LEVEL_SWITCH==1)
    goto('ALLDONE');
    return % Part of the goto functionality in Matlab, ignore
end
% IF WATER IS BELOW MINIMUM AND VALVE IS ON ||
VALVE = 0;
COUNTER1(2) = 0; % COUNTER1.CU
% ALLDONE LABEL'S COMMENT:
% LABEL ALLDONE
COUNTER1(5)=EXTERNAL_COUNTER_RESET;COUNTER1(4)=8;
COUNTER1=COUNTER_UP(COUNTER1);ALARM_COUNTER=COUNTER1(3);
TIMER1(2)=COUNTER1(6);TIMER1(4)=1;TIMER1=TIMER_ON(CLOCK_IN,TIMER1); %
COUNTER1.Q
BUZZ_SIGNAL_ALARM = TIMER1(6); % COUNTER1.QTIMER1.Q
LIGHT_SIGNAL_ALARM = COUNTER1(6); % COUNTER1.Q

%Output generation
OUTPUT_VECTOR=[ALARM_COUNTER*1,BUZZ_SIGNAL_ALARM*1,LIGHT_SIGNAL_ALARM*
1,
VALVE*1];
```

Anexo XVII

Listagem do código de controlo da serra traduzido para IL

```
PROGRAM STL_code_saw (*Original filename: STL code saw.stl*)
(*Translated by IEC 61131-3 IL Translator by Andre' Pereira*)
(*04-07-2011 12:55:54 Hora padrão de GMT*)
(*Dictionary used: Siemens Step 7 S-200 STL version: 0.5*)

VAR_GLOBAL
M0_5 : BOOL;
M0_0 : BOOL:=1;
M0_1 : BOOL;
M0_2 : BOOL;
M0_3 : BOOL;
M0_4 : BOOL;
M0_5  : BOOL;
END_VAR

VAR_INPUT
I0_3 : BOOL;
I0_0 : BOOL;
I0_5 : BOOL;
I0_1 : BOOL;
I0_2 : BOOL;
I0_4 : BOOL;
END_VAR


VAR_OUTPUT
Q0_1   : BOOL;
Q0_2 : BOOL;
Q0_3 : BOOL;
Q0_0  : BOOL;
Q0_4  : BOOL;
END_VAR

(* Siemens Step 7 S-200 STL translation dictionary *)

LD M0_5 (* START PROGRAM *)
AND I0_3
OR( M0_0
ANDN M0_1
)
ST M0_0
LD M0_0
AND I0_0
AND I0_5
AND I0_3
OR( M0_1
ANDN M0_2
)
ST M0_1
LD M0_1
AND I0_1
OR( M0_2
ANDN M0_3
)
ST M0_2
LD M0_2
AND I0_2
OR( M0_3
ANDN M0_4
)
ST M0_3
LD M0_3
AND I0_4
OR( M0_4
ANDN M0_5
)
ST M0_4
LD M0_4
AND I0_5
OR( M0_5
ANDN M0_0
)
ST M0_5  (*  END GRAFCET *)
(* WRITE THE OUTPUTS *)
LD M0_1
OR( M0_2
)
ST Q0_1   (* GO RIGHT *)
LD M0_2
ST Q0_2 (* SPIN SAW *)
LD M0_3
ST Q0_3
LD M0_4
ST Q0_0  (* GO LEFT *)
LD M0_5
ST Q0_4  (* LOWER MACHINE *)
(* END OF PROGRAM *)

END_PROGRAM
```

Anexo XVIII

Listagem do código de controlo da serra convertido para *Matlab*

```
function
[OUTPUT_VECTOR]=STL_CODE_SAW(CLOCK_IN,I0_0,I0_1,I0_2,I0_3,I0_4,I0_5)
%ORIGINAL FILENAME: STL CODE SAW.STL

%Converted by IEC-Matlab converter by Andre' Pereira
%04-07-2011 13:15:42 Hora padrÃ£o de GMT


global M0_0 M0_1 M0_2 M0_3 M0_4 M0_5 M0_5 Q0_0 Q0_1 Q0_2 Q0_3 Q0_4



% TRANSLATED BY IEC 61131-3 IL TRANSLATOR BY ANDRE' PEREIRA
% 04-07-2011 12:55:54 HORA PADRÃƒO DE GMT
% DICTIONARY USED: SIEMENS STEP 7 S-200 STL VERSION: 0.5
%  SIEMENS STEP 7 S-200 STL TRANSLATION DICTIONARY
M0_0 = ((M0_5&I0_3)|(M0_0&~M0_1)); %  START PROGRAM  ||
M0_1 = ((((M0_0&I0_0)&I0_5)&I0_3)|(M0_1&~M0_2));
M0_2 = ((M0_1&I0_1)|(M0_2&~M0_3));
M0_3 = ((M0_2&I0_2)|(M0_3&~M0_4));
M0_4 = ((M0_3&I0_4)|(M0_4&~M0_5));
M0_5 = ((M0_4&I0_5)|(M0_5&~M0_0)); %   END GRAFCET  ||
%  WRITE THE OUTPUTS  ||
Q0_1 = M0_1|(M0_2); %  GO RIGHT  ||
Q0_2 = M0_2; %  SPIN SAW  ||
Q0_3 = M0_3;
Q0_0 = M0_4; %  GO LEFT  ||
Q0_4 = M0_5; %  LOWER MACHINE  ||
%  END OF PROGRAM  ||

%Output generation
OUTPUT_VECTOR=[Q0_0*1,Q0_1*1,Q0_2*1,Q0_3*1,Q0_4*1];
```